Abstract Algebraic Effects via Syntactic Embeddings

ANONYMOUS AUTHOR(S)

Algebraic effect and handlers has been widely considered a versatile and modular abstraction for modeling user-defined effects. While there are many existing systems with algebraic effects, they are rarely designed with scalability in mind. In this paper, we design an effect system around the idea of making effect handlers scalable by providing a mechanism for effect abstraction. The paper can be divided into two parts. The first part of the paper presents a calculus that supports abstract algebraic effects by leveraging the method of syntactic embeddings. The second part presents a design of a high-level language with existential types. We give a formalization of the calculus and provide proofs for type soundness.

CCS Concepts: • **Theory of computation** → *Operational semantics*; *Abstraction*;

Additional Key Words and Phrases: algebraic effects, effect system, abstraction

ACM Reference Format:
Anonymous Author(s). 2020. Abstract Algebraic Effects via Syntactic Embeddings. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (January 2020), 29 pages.

1 INTRODUCTION

Algebraic effects (introduced by Plotkin and Power [2003]) and handlers (introduced by Plotkin and Pretnar [2009]) are an approach to computational effects based on a premise that impure behavior arises from a set of operations, and are recently gaining popularity due to their ability to model various form of computational effects such as exceptions, mutable states, async-await, etc.

Modularity is a key concept that separates abstract algebraic effects from the traditional way of using monad to model effects in purely functional programming [Schrijvers et al. 2019]. However, similar to the restrictive strand of work on effects, few works on algebraic effects have investigated the algebraic effect system on a larger scale, where abstraction between program components is important.

Abstract algebraic effects are first studied in Biernacki et al. [2019]. Similar to abstract types, abstract algebraic effects allow program components to define abstract effect signatures that are opaque to other components in the system. The difference between concrete and abstract effect signatures lies in the ability of program components to handle them. If an effect signature is concrete to a program component, then the operations are accessible to the component, and a handler can handle the effect by handling the operations in the signature. On the other hand, if an effect signature is abstract to one program component, then the component should not observe the operations defined in the effect signature, and is therefore unable to handle the effect. As abstraction is an important issue for module systems because it provides a separation of implementation details of functions from the interface, the abstraction of algebraic effects provides a similar benefit for modularity because it helps separate the component operations from the effect signature, ensuring that the client can only use the handler provided by the library to handle the effect.

1.1 Abstract Algebraic Effect Basics

We will use a high-level, simplified, object-oriented language to present following motivating examples. In this language, a module can contain effect declarations and method declarations. Effects can be defined as a collection of operations. For example, in figure 1, the effect `Nondet` is defined as an effect signature that contains a single operations `flip() : Bool`, which receives a unit

2020. 0164-0925/2020/1-ART1 $15.00
https://doi.org/

```
effect Nondet {
    flip(): Bool
}

type M
  effect E
  def mflip() : {this.E} Bool
  def handler(Unit -> {this.E} Bool) : {} Bool

module m: M
  effect E = {Nondet}
  def mflip() : {this.E} Unit
    flip()
  def handler(c: Unit -> {this.E} Bool) : {} Bool =
    handle c() with
    | flip() -> resume true
    | return x -> x
```

Fig. 1. Definition of Module m that Abstracts the Nondet Effect

```
m.handler(
  () => handle m.mflip() with
        | flip() -> resume false
        | return x -> x
)
```

Fig. 2. A Client of Module m

value and returns a boolean. On the other hand, effects can be defined as a collection of a set of effects. For example, in the definition of module `m` in figure `nondet`, the effect `E` is defined as an effect set that contains the `Nondet` effect. Module types contain declarations for both effects and methods. In figure 1, the type `M` declares an effect `E` and two methods `mflip` and `handler`. Note that the effect `E` is declared as an abstract effect as the type `M` does not give definition for it.

The definitions in figure 1 is a motivating example similar to an example in [Biernacki et al. 2019] that illustrates the challenges of implementing abstract algebraic effects. `Nondet` is a globally defined effect signature with an operation `flip` that returns a value of type `Bool`. Then we define a module `m` with type `M` with an abstract effect `E`, a method `mflip`, and a handler method `handler`. The effect `E` is defined as an alias of the effect `Nondet`, but is opaque to the outside world of the module, because `E` is defined as an abstract effect in the type `M`. The method `mflip` simply calls the `flip` operation, and the `handle` method handles the `flip` operation by returning `true`.

Figure 2 shows a client code of module `m` that calls the method `m.handler` and pass in a lambda expression that receives a unit value and then calls `m.mflip` within another handler that handles the `flip` operation. Since the effect of the method `m.mflip` is abstract, the inner handler should not handle the operation inside `m.mflip`. Instead, the operation should be handled by the outer handler method `m.handle`. If the method `m.mflip` is incorrectly handled by the inner handler, the result of the client computation would be the value `false`, which is incorrect.

As we can see, the abstraction of effect signatures differs from type abstractions, since the erasure of type information would make the abstraction unsound. So we need a language that keeps track of the information on effect abstraction during the evaluation of the program. In this paper, we leverage the method of syntactic type abstraction introduced by Grossman et al. [2000], who use

the notion of principals to track the flow of values with abstract types during the evaluation of a program.

2 AN AGENT-BASED CALCULUS

2.1 Motivating Example

Consider the simple case where we only have two agents, namely the client `c`, and the host `h`. And the host `h` defines an abstract effect `E`, exports a method that causes the effect, and a method that handles the effect.

```
module h =
  effect E = ...
  val m : 1 -> {E} 1 = ..
  ...
```

Now consider a client code that handles the `m` function from h

```
handle
  [m ()]^E_h
with
  op (x) -> ...
  return x -> ...
```

Since function `m` is called by the client code, we use a language construct called embedding to encapsulate the function call. The subscript `h` of the embedding indicates that the code inside the embedding is the host code, and the superscript `E` indicates that the effect of the code is `E`, which is abstract to the client. So the embedding ensures that the client would not be able to handle the operation inside the function `m`, therefore keeping the effect abstraction safe.

Besides making the client unable to handle an abstract effect, we need to make sure that a host code can "rediscover" the effect exported by itself. The scenario would be exhibited by the following host code. The client code we showed earlier is now embedded into a host handler that handles the effect E. Because the outer-most handler is now in host code, it would be able to handle the effect operation inside the function `m`.

```
handle
  [handle
    [m ()]^E_h
  with
    op (x) -> ...
    return x -> ...]^E_c
with
  op (x) -> ...
  return x -> ...
```

2.2 Design of the Core Calculus

The core calculus is based on standard call-by-value lambda calculus that is extended with effect handlers. We introduce the basic design of the calculus in this section. A full formalization of the calculus is presented in section 3.

2.2.1 Terms. Terms are divided into expressions and computations. Expressions are pure terms that do not cause effects, while computations might cause effects. Similar to regular designs of lambda calculus, expressions include unit value `()`, or a lambda expression $\lambda x : \tau.\ c$.

Computations are terms that might cause effects. A pure expression can be lifted to a computation by the `return` keyword. For example, `return ()` is a computation that simply returns a unit value.

Some computations are related to algebraic effects, such as the operation call `op(e; y.c)` and effect handler `handle c with h`. In the operation call `op(e; y.c)`, `op` is the name of the operation, `e` is the argument for the operation call, and `y.c` is the continuation of the operation call, where variable `y` is bound to the result of the operation call. In an effect handler `handle c with h`, `c` can be an arbitrary computation, and `h` contains handling clause for operations. For example,

```
handle
  flip(); y. if (y) return 0 else return 1)
with
  | flip(x; k) -> k true
  | return x -> return x
```

The flip operation is handled by a handler that contains two clauses. In the first clause `flip(x; k) -> k true`, x is bound to the input argument, which in this case is the unit variable. k is the continuation of the operation call, which in this case would be $\lambda y : bool.$ `if (y) return 0 else return 1`. Since the value *true* is passed into the continuation k, the result of the computation would be `return 0`.

2.2.2 Types. Similar to terms, types are also split into expression types and computation types. We generally use τ to represent expression type and σ to represent computation types. An expression type is either a unit type 1 or an arrow type $\tau \longrightarrow \sigma$, where τ is another expression type and σ is a computation type.

A computation type $\{\varepsilon\}\tau$ comprises two parts: an effect set ε and a return value type τ. The effect set ε can either be an empty set $\cdot$, or an effect label f attached to another effect set f, ε. An effect label can be an abstract label, or a concrete operation name such as `flip`. The return value type τ is the type of the computation result. For example, the computation `return ()` has type $\{\}1$, because it does not cause an effect and return a unit value. On the other hand, the computation `flip(); y.return 0)` has type $\{\texttt{flip}\}Int$, because it causes the flip effect and returns an integer.

2.2.3 Agents and Embeddings. We introduce agents to track the effect abstraction information during the evaluation of programs. An agent could be considered a module with its own knowledge of effect abstraction. For example in section 2.1, the agent `h` has the knowledge that $E = op$, but the agent `c` is oblivious to the definition of E, and this distinction affects their ability to handle the effect. Every term in our calculus is associated to a specific agent.

Code that belongs to different agents can interact with each other though embeddings. Assuming that we have two different agents i and j, an embedded expression $[e_j]_j^\tau$ would be an expression of agent i, where expression e_j is an expression of agent j, and the type annotation τ is the type of e_j in agent i's perspective. For example, the expression

$$[\lambda x : 1.\ ()]_j^{1\to\{\}1}$$

is annotated with type $1 \longrightarrow \{\}1$ because the embedded expression $\lambda x : 1.()$ is a unit function with no effect.

Computations can be embedded in a similar fashion. An embedded computation $[c_j]_j^\sigma$ is a computation of agent i, where the computation type σ is the type annotation for the embedded computation c_j. For example, assuming that agent j defines the effect `Nondet` that contains operation `flip`, a call to the flip operation by agent j can be embedded as

$$[flip(); y.\texttt{return}\ y)]_j^{\{Nondet\}Bool}$$

which is a term of agent i. The type annotation is $\{Nondet\}Bool$ because the operation `flip` is called, and `Nondet` is the effect signature that contains `flip`. Moreover, the return type of the embedded computation is boolean, so the `Bool` is annotated as the return type.

2.3 Running Example in the Core Calculus

In this section we show the process of evaluation of the example program presented in figure 1 and figure 2. The original example could be simplified as following:

```
op : 1 -> 1

type B
  effect f // f is an abstract effect
  def m() : {f} Unit
  def handler(c: 1 -> {f} 1) : {} Int

module b: B
  effect f = {op}
  def m() : {f} Unit
    op ()
  def handler(c: 1 -> {f} 1) : {} Int
    handle c () with
    | op _ -> return 1
    | return _ -> return 0
```

The operation `op` is defined globally, and the module `b` defines effect `f` as an effect signature that contains the operation `op`, an effectful method `m`, and an handler method `handler`. The example program that we will evaluate is written as following. The handler method from module `b` is invoked, and the argument is a computation that calls the method `b.m`, which is surrounded by a handler that handles `op`. From the perspective of the client, the effect of `b.m()` shouldn't be handled by the inner handler because the effect `f` is abstract to client.

```
b.handler(
  () => handle b.m() with
          | op _ -> resume ()
          | return _ -> ()
 )
```

Then we rewrite the example program in our agent-based calculus: Since the code is a client of module `b`, any method call to a member of the module `b` should be surrounded by an embedding. So the handler functions is wrapped in an embedding with type annotation `(1 -> {f} -> 1) -> {} Int`, and the handled function in the argument is also embedded with an annotation `1 -> {f} 1`. In the following evaluation process we assume that there is an agent `b` that represents the module `b`, and an agent `a` that represents the client code that invokes functions from module `b`

```
// Translation of Example Program
[λc: 1 -> {f} 1.
  handle c() with
  | op(x, k) -> return 1
  | return x -> return 0)]_b^{(1→{f}1)→{}int}
(λ_:1.
  handle [λ_:1. op()]_b^{1→{f}1} () with
  | op(x, k) -> k ()
  | return _ -> return ())
```

Then we evaluate the handler function so that the binder `λc : 1 -> {f} 1` is lifted out of the embedding, because embeddings cannot be a normal form of an expression. The detail on how we evaluate an embedded expression could be found in section 3.3. The program is therefore evaluated to

```
(λc: 1 -> {f} 1.
  [handle [c]_a^{1->{f}1} () with
  | op(x, k) -> return 1
  | return x -> return 0)]_b^{}int)
(λ_:1.
  handle [λ_:1. op()]_b^{1->{f}1} () with
  | op(x, k) -> k ()
  | return _ -> return ())
```

Since the handler function is evaluated to a lambda expression, we can perform a β-reduction:

```
 [handle
   [λ_:1.
      handle [λ_:1. op()]_b^{1->{f}1} () with
      | op(x, k) -> k ()
      | return _ -> return () ]_a^{1->{f}1} ()
  with
  | op(x, k) -> return 1
  | return x -> return 0)]_b^{}int
```

Then we evaluate the computation handled by the outermost handler. Specifically, we pass the unit value () into the first lambda expression.

```
[handle
    [handle [λ_:1. op()]_b^{1->{f}1} () with
      | op(x, k) -> k ()
      | return _ -> return () ]_a^{f}1
  with
  | op(x, k) -> return 1
  | return x -> return 0)]_b^{}int
```

Now we have eliminated all lambda expressions and have two nested handlers and three nested embeddings. The inner most embedding `[λ_:1. op()]`$_b^{1\to\{f\}1}$ embeds the function call from the module `b`. The inner handler that handles the function call is a handler provided by the client `a`, and is therefore embedded by an embedding with subscript `a`. Recall from our discussion in the beginning of section 2.3, this handler should not handle the operation `op` because agent `a` is oblivious to the definition of effect `f`. Since we know syntatically that the inner handler is from agent `a`, this handler would be skipped. On the other hand, the outer handler is annotated with subscript `b`, which indicates that it is provided by agent `b`. Because agent `b` itself should be able to handle the its own effect `f`, the effect of the function call would be handled by the outer handler.

3 FORMALIZATION OF THE AGENT-BASED CALCULUS

3.1 Syntax

This section describes a variant of the simply typed lambda calculus that maintains a syntactic distinction between agents during evaluation. Figure 3 gives the syntax of our calculus. As our previous discussion, it is crucial to keep track of the effect abstraction information during the evaluation of the program. It is therefore natural to divide the code into agents and allow each agent to export abstract effect signatures. We assume that there are n agents, and variables i, j, k range over the set of agents.

Every term in this language is assigned to an agent. And terms are split into inert expressions and potentially effectful computations, following an approach called *fine-grain call-by-value*, introduced

$$
\begin{array}{ll}
\text{(agents)} & i, j ::= \{1 \ldots n\} \\
\text{(lists)} & l ::= i \mid il \\
\text{(value types)} & \tau ::= 1 \mid \tau \to \sigma \\
\text{(computation types)} & \sigma ::= \{\varepsilon\}\tau \\
\text{(effect types)} & \varepsilon ::= \cdot \mid f, \varepsilon \mid op, \varepsilon \\
\text{(i-values)} & v_i ::= ()_i \mid \lambda x_i : \tau.\ c_i \\
\text{(i-expressions)} & e_i ::= x_i \mid v_i \mid [e_j]^{\tau}_{j} \\
\text{(i-computations)} & c_i ::= \texttt{return}\ e_i \mid op(e_i, y.c_i) \mid \texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i' \mid e_i\ e_i' \\
 & \quad \mid \texttt{with}\ h_i\ \texttt{handle}\ c_i \mid\ [c_j]^{\sigma}_{j} \mid [op]^{\varepsilon}_{l}(e_i, y_i.c_i) \\
\text{(i-handlers)} & h_i ::= \texttt{handler}\ \{\texttt{return}\ x_i \mapsto c_i^r, op^1(x_i^1, k^1) \mapsto c_i^1, \ldots, op^n(x_i^n, k^n) \mapsto c_i^n\}
\end{array}
$$

Fig. 3. Syntax for multi-agent calculus

by Levy et al. [2003]. We use the notation *i-expression* and *i-computation* to denote expressions and computations in the agent i. We use subscripts to indicate a term is assigned to an agent, however, we will omit the subscript if the agent the term belongs to is not important or obvious in the context.

An *i-value* is an *i-expression* that cannot be further reduced. There are two forms of *i-value*: the unit $()$, and the lambda abstraction $\lambda x_i : \tau.\ c_i$. *i-expressions* include variable x_i, value v_i, and embedded expressions $[e_j]^{\tau}_{j}$. *i-computations* are the terms that can potentially cause effects, and consists of return statement $\texttt{return}\ e_i$, operation call $op(e_i; y_i.c_i)$, sequencing $\texttt{do}\ x_i \leftarrow c_i\ \texttt{in}\ c_i'$, application $e_i\ e_i'$, handling $\texttt{with}\ h_i\ \texttt{handle}\ c_i$, embedded computation $[c_j]^{\sigma}_{j}$, and embedded operation call $[op]^{\varepsilon}_{l}(e_i; y_i.c_i)$. There are a few things worth mentioning:

Sequencing: In $\texttt{do}\ x \leftarrow c\ \texttt{in}\ c'$, we first evaluate c, bind the return value of c' to x and then evluate c_2

Operation Calls: The call $op(e; y.c)$ passes the parameter e to the operation op, binds the return value of the operation call to y, and continue by evaluating the computation c. Note that the encompassing handler could potentially change the behavior of the operation. Explicit continuations greatly simplifies the operational semantics of the language, because continuations make the order of the execution of operations explicit.

Embeddings: the term $[e_j]^{\tau}_{j}$ is an i-expression, where e_j is an embedded j-expression. The type τ is exported by the agent j as the type of the embedded expression. Similarly, $[c_j]^{\sigma}_{j}$ is an embedded j-computation with exported type σ.

Embedded Operations: The embedded operation $[op]^{\varepsilon}_{l}(e; y.c)$ is an operation call that is annotated with effect ε. l is a list of agents that have contributed to the formation of the annotation. The embedded operations should not appear in the source code, as they are an intermediate form of computation that keeps track of the effect annotation of operations. More details of this construct are given in section 3.3 on Operational Semantics.

Similar to terms, types are also divided into expression types and computation types. There are two forms of expression types τ: the unit type 1, and the arrow type $\tau \to \sigma$. As for the computation type σ, there is only one form: $\{\varepsilon\}\tau$, where ε is a set of effects that the computation might induce, and τ is the type of the return value of the computation.

The effect type ε represents an unordered set of effects that can be empty $\cdot$. A effect type can be extended by either an effect label f, or an operation op.

The i-handler h_i must contain a return clause `return` $x \mapsto c$, which handles the case when the handled computation directly returns a value. The returned value is bound to the variable x, and the entire handling computation evaluates to the computation c. A handler may also contain clauses that handle operations. For example, the clause $op(x, k) \mapsto c$ handles the operation op. More details can be found in the dynamic semantics section.

3.2 Agent-Specific Type Information

We use agents to model a module system where each module can have private information about effect abstraction. Each agent in our language has limited knowledge of effect abstraction. For example, an agent i might knows that effect `Nondet = flip(): Bool`, and an agent j does not have this information. As a result, agent i would be able to handle a computation with effect `Nondet`, while the agent j would not be able to handle the effect, because from agent j's point of view, the effect `Nondet` is an abstract label without an operation. Furthermore, we need to ensure the consistency of the information on effect abstraction, that is, agent j should not think that the effect `Nondet = read(): String`, which would contradict with the knowledge of agent i.

The model of effect abstraction information is similar to the model of type information in [Grossman et al. 2000]. To capture effect abstraction information, each agent i has a partial function δ_i that maps an effect label to an effect type. There are two requirements for these maps: (1) For each effect label f, the if there are two agents that knows the implementation of the effect f, then their knowledge about the implementation must be the same. (2) For each effect label f, there is a unique and most concrete interpretation of f. We would not allow the effect label f itself to appear in the implementation of f. Examples like $\delta_i(f) = \{f\}$ and $\delta_i(f) = \{f, op\}$ would be rejected.

Definition 1. A set $\{\delta_1, \ldots, \delta_n\}$ of maps from effect labels to effects is compatible if

(1) For all $i, j \in 1 \ldots n$ if $f \in Dom(\delta_i) \cap Dom(\delta_j)$, then $\delta_i(f) = \delta_i(j)$.
(2) Effect labels can be totally ordered such that for every agent i and effect label f, all effect labels in $\delta_i(f)$ precede f.

Then we define the a total function Δ_i that refines an effect type:

Definition 2. Δ_i is a function that maps an effect type to another effect type, using the effect abstraction knowledge of the agent i.

$$\Delta_i(\cdot) = \cdot$$
$$\Delta_i(op, \varepsilon) = op, \Delta_i(\varepsilon)$$
$$\Delta_i(f, \varepsilon) = \begin{cases} f, \Delta_i(\varepsilon) & \text{if } f \notin Dom(\delta_i) \\ \varepsilon', \Delta_i(\varepsilon) & \text{if } \delta_i(f) = \varepsilon' \end{cases}$$

The definition of compatibility ensures that there is a fixpoint for repeatedly refining an effect label ε using the function Δ_i. We call such fixpoint $\overline{\Delta_i}(\varepsilon)$.

Definition 3. $\overline{\Delta_i}(\varepsilon') = \varepsilon$ if there is some $n \geq 0$

$$\underbrace{\Delta_i(\ldots(\Delta_i(\varepsilon')\ldots)}_{\text{n applications}} = \underbrace{\Delta_i(\ldots(\Delta_i(\varepsilon')\ldots)}_{\text{n+1 applications}} = \varepsilon$$

To see how we apply Δ to reach a fix point, consider two effect labels f and g and an agent i. Agent i knows that the effect f is implemented by two operations op_1 and op_2. Then consider applying Δ_i to the effect type f, g. We get $\Delta_i(f, g) = op_1, op_2, g$. Since op_1, op_2, and label g is the most concrete form of effects, Δ_i cannot further refine the resulting type, so we have reached the

fix point. So we have $\overline{\Delta_i}(f, g) = op_1, op_2, g$. As we can see, by repeatedly applying Δ_i and getting to a fixpoint, we effectively collect all operations and abstract effect labels in the agent i's perspective.

We assume that the type information for operations is public to all agents. The type for an operation op is contained by a separate map Σ, which maps an operation op to an arrow type $\tau_A \longrightarrow \tau_B$. Note that this is different from the function type in our calculus, which has the form $\tau \longrightarrow \sigma$.

3.3 Operational Semantics

$\boxed{e \longrightarrow e'}$

$$\frac{e_j \mapsto e'_j}{[e_j]^\tau_j \longrightarrow [e'_j]^\tau_j}\ \text{(E-Congruence)} \qquad \frac{}{[()_j]^1_j \longrightarrow ()_i}\ \text{(E-Unit)}$$

$$\frac{}{[\lambda x_j : \tau'.\, c_j]^{\tau\to\sigma}_j \longrightarrow \lambda x_i : \tau.\, [\{[x_i]^{\tau'}_i / x_j\} c_j]^\sigma_j}\ \text{(E-Lambda)}$$

Fig. 4. Operational Semantics for Expressions

The reduction rules for terms are dependent on the agent of the terms. Figure 4 shows that operational semantics for expressions of agent i. (E-Congruence) shows that a j-expression embedded agent-i should be evaluated using the reduction rules for agent j first. The (E-Unit) and (E-Lambda) rules show that we can lift an embedded j-value to agent i, so the value becomes an i-value. The (E-Unit) rule simply lifts the unit value out of the embedding. The (E-Lambda) rule is more interesting: The value embedded is a lambda expression of agent j. We lift the argument out of the embedding. However, the type annotating the argument is changed from τ' to the exported argument type τ, because the reduced expression should have the exported type $\tau \longrightarrow \sigma$. The body of the reduced expression is an embedded j-computation, so the variable x_i should be encapsulated by an embedding, because any i-term should be embedded in a j-term. We annotate x_i with type τ' because the original lambda function expects a value of type τ'.

Figure 5 shows the reduction rules for i-computations. (E-Ret) is the congruence rule that evaluates the expression in a return statement. (E-Op) evaluates the input argument for the operation call. Note that there is no non-congruence reduction rule for operation calls because the semantics for operations are defined by the handler encapsulating it.

(E-Embed1) is the congruence rule for embedded computations. (E-Embed2) lifts the return statement out of the embedding. We can safely remove the effect annotation ε because the statement that returns a value v_j cannot cause any effect.

(E-Embed3) lifts an operation call out of the embedding. This rule introduces embedded operation as a new language construct. We annotate the embedded operation with the effect annotation of the whole computation. Since the argument value for op is a j-value, we need to embed it as an i-value, and annotate it with type τ_A. The continuation c_j is still embedded, and we substitute the embedded variable y_i for y_j. y_i should be embedded because i-values should be embedded in a j-value.

The (E-Embed4) rule is very similar to (E-Embed3) with one difference being that op is already embedded. In this case, we override the annotation on op with the annotation for the whole computation and update the agent list in the subscript of the operation. We add j to the agent list because the agent j has contributed to the effect annotation of the operation.

(E-EmbedOp1) evaluates the argument of an embedded operation. (E-EmbedOp2) refines the effect annotation of an operation by looking up the effect ε from the type information of the agent i, Δ_i. (E-EmbedOp3) lifts the operation out of an embedding when the annotation contains

$\boxed{c \longrightarrow c'}$

$$\frac{e_i \mapsto e_i'}{\texttt{return}\ e_i \longrightarrow \texttt{return}\ e_i'}\ (\textsc{E-Ret}) \qquad \frac{e_i \mapsto e_i'}{op(e_i, y_i.c_i) \longrightarrow op(e_i', y_i, c_i)}\ (\textsc{E-Op})$$

$$\frac{c_j \longrightarrow c_j'}{[c_j]_l^\sigma \longrightarrow [c_j']_l^\sigma}\ (\textsc{E-Embed1}) \qquad \frac{}{[\ \texttt{return}\ v_j]_l^{\{\varepsilon\}\tau} \longrightarrow\ \texttt{return}\ [v_j]_l^\tau}\ (\textsc{E-Embed2})$$

$$\frac{\Sigma(op) = \tau_A \to \tau_B}{[op_j(v_j; y_j.c_j)]_j^{\{\varepsilon\}\tau} \longrightarrow [op_j]_j^\varepsilon([v_j]_j^{\tau_A}; y_i.\{[y_i]_i^{\tau_B}/y_j\}[c_j]_j^{\{\varepsilon\}\tau})}\ (\textsc{E-Embed3})$$

$$\frac{\Sigma(op) = \tau_A \to \tau_B \quad \Delta_j(\varepsilon') = \varepsilon' \quad op \notin \varepsilon'}{[[op_k]_l^{\varepsilon'}(v_j; y_j.c_j)]_j^{\{\varepsilon\}\tau} \longrightarrow [op_k]_{lj}^\varepsilon([v_j]_j^{\tau_A}; y_i.\{[y_i]_i^{\tau_B}/y_j\}[c_j]_j^{\{\varepsilon\}\tau})}\ (\textsc{E-Embed4})$$

$$\frac{e_i \longrightarrow e_i'}{[op]_l^\varepsilon(e_i; y_i.c_i) \longrightarrow [op]_l^\varepsilon(e_i'; y_i.c_i)}\ (\textsc{E-EmbedOp1}) \qquad \frac{\overline{\Delta_i}(\varepsilon) = \varepsilon'}{[op]_l^\varepsilon(v_i; y_i.c_i) \longrightarrow [op]_l^{\varepsilon'}(v_i; y_i.c_i)}\ (\textsc{E-EmbedOp2})$$

$$\frac{\Delta_i(\varepsilon) = \varepsilon \quad op \in \varepsilon}{[op]_l^\varepsilon(v_i; y_i.c_i) \longrightarrow op(v_i; y_i.c_i)}\ (\textsc{E-EmbedOp3}) \qquad \frac{\Delta_i(\varepsilon) = \varepsilon \quad op \notin \varepsilon \quad op' \in \varepsilon}{[op]_l^\varepsilon(v_i; y_i.c_i) \longrightarrow [op]^{\varepsilon\setminus op'}(v_i; y_i.c_i)}\ (\textsc{E-EmbedOp4})$$

$$\frac{e_i \longrightarrow e_i''}{e_i\ e_i' \longrightarrow e_i''\ e_i'}\ (\textsc{E-App1}) \qquad \frac{e_i \longrightarrow e_i'}{v_i\ e_i \longrightarrow v_i\ e_i'}\ (\textsc{E-App2}) \qquad \frac{}{(\lambda x_i : \tau.\ c_i)\ v_i \longrightarrow \{v_i/x_i\}c_i}\ (\textsc{E-App3})$$

$$\frac{c_i \longrightarrow c_i''}{\texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i' \longrightarrow \texttt{do}\ x \leftarrow c_i''\ \texttt{in} c_i'}\ (\textsc{E-Seq1}) \qquad \frac{}{\texttt{do}\ x \leftarrow \texttt{return}\ v_i\ \texttt{in}\ c_i' \longrightarrow \{v_i/x\}c_i'}\ (\textsc{E-Seq2})$$

$$\frac{}{\texttt{do}\ x \leftarrow op_i(v_i; y_i.c_i)\texttt{in}\ c_i' \longrightarrow op_i(v_i; y_i.\ \texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i')}\ (\textsc{E-Seq3})$$

$$\frac{\Delta_i(\varepsilon) = \varepsilon \quad op \notin \varepsilon}{\texttt{do}\ x \leftarrow [op_j]_l^\varepsilon(v_i; y_i.c_i)\texttt{in}\ c_i' \longrightarrow [op_j]_l^\varepsilon(v_i; y_i.\ \texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i')}\ (\textsc{E-Seq4})$$

$$\frac{c_i \longrightarrow c_i'}{\texttt{with}\ h_i\ \texttt{handle}\ c_i \longrightarrow\ \texttt{with}\ h_i\ \texttt{handle}\ c_i'}\ (\textsc{E-Handle1})$$

$$\frac{\texttt{return}\ x_i \mapsto c_i' \in h_i}{\texttt{with}\ h_i\ \texttt{handle}\ \texttt{return}\ v_i \longrightarrow \{v_i/x_i\}c_i'}\ (\textsc{E-Handle2})$$

$$\frac{op(x_i; k) \mapsto c_i' \in h_i \quad \Sigma(op) = \tau_A \to \tau_B}{\texttt{with}\ h_i\ \texttt{handle}\ op(v_i y_i.c_i) \longrightarrow \{v_i/x_i\}\{(\lambda y_i : \tau_B.\ \texttt{with}\ h_i\ \texttt{handle}\ c_i)/k\}c_i'}\ (\textsc{E-Handle3})$$

$$\frac{op(x_i; k) \mapsto c_i' \notin h_i}{\texttt{with}\ h_i\ \texttt{handle}\ op(v_i, y_i.c_i) \longrightarrow op(v_i; y_i.\ \texttt{with}\ h_i\ \texttt{handle}\ c_i))}\ (\textsc{E-Handle4})$$

$$\frac{\Delta_i(\varepsilon) = \varepsilon \quad op \notin \varepsilon}{\texttt{with}\ h_i\ \texttt{handle}\ [op]_l^\varepsilon(v_i, y_i.c_i) \longrightarrow [op]_l^\varepsilon(v_i; y_i.\ \texttt{with}\ h_i\ \texttt{handle}\ c_i))}\ (\textsc{E-Handle5})$$

Fig. 5. Operational Semantics for Computations

the operation, because the agent i has enough information about effect abstraction to handle the operation. Note that in the premise, we require that the effect annotation cannot be further refined, in order to ensure determinism of evaluation. (E-EmbedOp4) removes an operation that is not op

$\boxed{\Gamma \vdash e_i : \tau}$

$$\frac{}{\Gamma \vdash ()_i : 1}\ (\text{T-UNIT}) \qquad \frac{}{\Gamma \vdash x_i : \Gamma(x_i)}\ (\text{T-VAR}) \qquad \frac{\Gamma, x_i : \tau \vdash c_i : \sigma}{\Gamma \vdash (\lambda x_i : \tau.\ c_i) : \tau \to \sigma}\ (\text{T-LAM})$$

$$\frac{\Gamma \vdash e_j : \tau' \quad \Gamma \vdash \tau' \leq_{ji} \tau}{\Gamma \vdash [e_j]^{\tau}_{j} : \tau}\ (\text{T-EMBEDEXP})$$

$\boxed{\Gamma \vdash c_i : \sigma}$

$$\frac{\Gamma \vdash e_i : \tau \quad \Delta_i(\varepsilon) = \varepsilon}{\Gamma \vdash \texttt{return}\ e_i : \{\varepsilon\}\tau}\ (\text{T-RET}) \qquad \frac{\Sigma(op) = \tau_A \to \tau_B \quad \Gamma \vdash e_i : \tau_A \quad \Gamma, y_i : \tau_B \vdash c_i : \{\varepsilon\}\tau \quad op \in \Delta_i(\varepsilon)}{\Gamma \vdash op(e_i; y_i.c_i) : \{\varepsilon\}\tau}\ (\text{T-OP})$$

$$\frac{\Gamma \vdash c_i : \{\varepsilon\}\tau \quad \Gamma, x_i : \tau \vdash c_i' : \{\varepsilon\}\tau'}{\Gamma \vdash \texttt{do}\ x_i \leftarrow c_i\ \texttt{in}\ c_i' : \{\varepsilon\}\tau'}\ (\text{T-SEQ}) \qquad \frac{\Gamma \vdash e_1 : \tau \to \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2\ : \sigma}\ (\text{T-APP})$$

$$\frac{\begin{array}{c} h_i = \{\ \texttt{return}\ x \mapsto c^r, op^1(x;k) \mapsto c^1, \ldots, op^n(x;k) \mapsto c^n\ \} \\ \Gamma, x : \tau_A \vdash c^r : \{\varepsilon'\}\tau_B \quad \left\{\Sigma(op^i) = \tau_i \to \tau_i' \quad \Gamma, x : \tau_i, k : \tau_i' \to \{\varepsilon'\}\tau_B \vdash c^i : \{\varepsilon'\}\tau_B\right\}_{1 \leq i \leq n} \\ \Gamma \vdash c_i : \{\varepsilon\}\tau_A \quad \varepsilon \setminus \{op^i\}_{1 \leq i \leq n} \subseteq \varepsilon' \end{array}}{\Gamma \vdash\ \texttt{with}\ h_i\ \texttt{handle}\ c_i : \{\varepsilon'\}\tau_B}\ (\text{T-HANDLE})$$

$$\frac{\Gamma \vdash c_j : \sigma' \quad \Gamma \vdash \sigma' \leq_{li} \sigma}{\Gamma \vdash [c_j]^{\sigma}_{l} : \sigma}\ (\text{T-EMBED})$$

$$\frac{\Sigma(op) = \tau_A \to \tau_B \quad \Gamma \vdash e_i : \tau_A \quad \Gamma, y_i : \tau_B \vdash c_i : \{\varepsilon'\}\tau \quad \overline{\Delta_i}(\varepsilon) \subseteq \overline{\Delta_i}(\varepsilon') \quad \Gamma \vdash op \leq_{li} \varepsilon}{\Gamma \vdash [op]^{\varepsilon}_{l}(e_i; y_i.c_i) : \{\overline{\Delta_i}(\varepsilon')\}\tau}\ (\text{T-EMBEDOP})$$

Fig. 6. Static Semantics

out of the effect annotation. This step does not affect the correctness of the type information, and is helpful in our proof of type soundness.

(E-App1), (E-App2) and (E-App3) are standard call-by-value semantics for applications. (E-Seq1) evaluates the first computation in a sequence of computations. (E-Seq2) binds the return value of the first computation to a variable in the second computation. (E-Seq3) witnesses an operation call as the first computation in a sequence. Since there is no way to further evaluate an operation right away, we propagate the operation call outwards and defer further evaluation to the continuation of the call. (E-Seq4) is similar to (E-Seq3), and requires that the effect annotation on the embedding to be the most concrete annotation.

(E-Handle1) simply evaluates the computation encapsulated by the handler. In (E-Handle2), the computation returns a value, so we substitute the value into the computation of the clause that handles the return statement in the handler. (E-Handle3) shows that case when the handler h_i has a matching clause for the operation op. We substitute the argument v_i for x_i, and substitute the continuation of the operation for k. The continuation function receives an argument of type τ_B, which is the result type of the operation op, and computes the continuation of the operation c_i, but encapsulates the computation with the handler h_i. The (E-Handle4) shows the case when the operation is not handled by the handler, so we propagate the operation outwards to wait for another handler to handle it. The (E-Handler5) ensures that abstracted effects are not handled: If the current agent cannot refine the effect annotation, then the operation is abstract and cannot be handled, and is therefore propagated outward.

3.4 Static Semantics

3.4.1 Typing Rules. Figure 6 shows the static semantics of the core-calculus. Static semantics includes typing rules for both expressions and computations. Note that the typing rules depend on the agent each expression or computation belongs to. We assume that the following rules assign types to terms of agent i.

The rule (T-Unit) assigns the unit type 1 to a unit value. (T-Var) looks up a type of a variable from the context. (T-Lam) is the standard rule for typing a lambda function. Note that the body of a lambda is computation, so we need to use the typing judgment for computation in the premise of this rule. (T-EmbedExp) assigns type to expression embeddings: The embedding has type τ if the embedded expression e_j is assigned to the type τ', and τ is related to τ' by the list of agents li. We will elaborate on type relations later.

(T-Ret) assigns a type to a return statement: As expected, the expression part of the computation type matches the type of the returned expression. However we can annotate the return statement with an arbitrary effect set, because our type system does not describe the precise effect in computations, but gives the upper bound of effect in computations.

(T-Op) shows the typing rule for operation calls. Again, since we allow effect annotations to be an upper bound on effect, we can require the operation op to be in the effect set ε.

(T-Handle) shows the typing rule for the effect handling statement `with` h_i `handle` c_i. c_i is a computation with effect type ε and return type τ_A. h_i is a handler that contains a clause that handles operations $op^1, \ldots, op^n$. For the return clause, given the type of variable x is τ_A, the type of c^r must be $\{\varepsilon'\}\tau_B$. For the clause handling the operation op^i, which has the operation type $\tau_i \to \tau_i'$, if variable x has type τ_i, and continuation has the type $\tau_i' \to \{\varepsilon'\}\tau_B$, then the handling computation c^i must have the type $\{\varepsilon'\}\tau_B$. The effect type after handling, ε' should contain all of the effects that are not handled by the handler.

(T-Embed) is very similar to (T-EmbedExp), where we use the type of the embedded computation and the type relation judgment to assign a type to the embedding.

(T-EmbedOp) first computes the type of c_i given that y_i has the correct type. It is required that the effect annotation on the embedding is a part of the effect of c_i. And op should be related to the effect annotation by the type relations.

$\boxed{\tau \leq_l \tau'}$

$$\frac{}{1 \leq_l 1}\ \text{(R-UNIT)} \qquad \frac{\tau \leq_l \tau' \quad \sigma \leq_l \sigma'}{\tau \to \sigma \leq_l \tau' \to \sigma'}\ \text{(R-ARROW)}$$

$\boxed{\sigma \leq_l \sigma}$

$$\frac{\varepsilon \leq_l \varepsilon' \quad \tau \leq_l \tau'}{\{\varepsilon\}\tau \leq_l \{\varepsilon'\}\tau'}\ \text{(R-SIGMA)}$$

$\boxed{\varepsilon \leq_l \varepsilon}$

$$\frac{\overline{\Delta_i}(\varepsilon) \subseteq \overline{\Delta_i}(\varepsilon')}{\varepsilon \leq_i \varepsilon'}\ \text{(R-EFF1)} \qquad \frac{\varepsilon \leq_l \varepsilon'' \quad \varepsilon'' \leq_{l'} \varepsilon'}{\varepsilon \leq_{ll'} \varepsilon'}\ \text{(R-EFF2)}$$

Fig. 7. Type Relations

3.4.2 Type Relations. Type relations ensure the soundness of abstraction. The goal of type relations is to prohibit embeddings from exporting incorrect effect abstractions. For example, if a i-computation uses an effect operation $flip : 1 \to bool$, which is an operation of effect `Nondet`, then whenever it is embedded in another agent, it should be annotated with effect `Nondet`, but should not be annotated with the empty effect or other effects that does not contain `Nondet`.

The judgements for expression types are of the form $\tau \leq_l \tau$, where l is a list of agents that provide the type abstraction information used by the relation. (R-Unit) shows that unit type relates to itself.

(R-Arrow) relates two arrow types given that the input types and the output types are related, To relate two computation types, we just need to ensure the effect types and return types are related.

The relation for effect types does the actual work. By (R-EFF1), two effect types are related under a single agent i if the first effect is a subset of the second effect after refinement by the type information provided by i. (R-EFF2) shows that by using type information from a list of agents, we can combine the chain of relation between effects.

3.5 Type Soundnesss

In this section, we state the standard type-safety theorems for the core calculus. Since we split terms into expressions and computations, we state progress and preservation lemmas separately. The proofs to theorems stated in this section can be found in appendix A.

We begin by stating the preservation and progress lemmas. The preservation lemma for expressions and computations are rather standard, except for the fact that we consider terms for each agent separately. In lemma 1, if an i-expression steps to another i-expression, then the new expression should have the identical type as the original expression. Similarly, in lemma 2, the new computation should have the same effect type and return type as the original computation.

Lemma 1 (Preservation for Expressions)**.**
For all agent i, if $\Gamma \vdash e_i : \tau$ and $e_i \mapsto e_i'$, then $\Gamma \vdash e_i' : \tau$.

Lemma 2 (Preservation for Computations)**.**
For all agent i, if $\Gamma \vdash c_i : \{\varepsilon\}\tau$ and $c_i \longrightarrow c_i'$, then $\Gamma \vdash c_i' : \{\varepsilon\}\tau$

The progress lemma for expressions (lemma 3) is also standard. If an i-expression is well-typed, then it either steps to another expression or is already a value.

Lemma 3 (Progress for Expressions)**.**
For agent i, if $\varnothing \vdash e_i : \tau$ then either $e_i = v_i$ or $e_i \longrightarrow e_i'$.

The progress lemma for computations (lemma 4) is a bit more involved. For any well-typed i-computation, there are four possibilities. Similar to expressions, a computation can evaluate to another computation. Otherwise, a computation could potentially be a return statement, an operation, or an embedded operation. The dynamic semantics ensures that these are the only possible final configurations for a computation.

Lemma 4 (Progress for Computations)**.**
If $\varnothing \vdash c_i : \{\varepsilon\}\tau$ then either
(1) $c_i \longrightarrow c_i'$
(2) $c_i = \texttt{return}\ v_i$
(3) $c_i = op(v_i; y_i.c_i')$
(4) $c_i = [op]_l^{\varepsilon}(v_i; y_i.c_i')$

4 ALGEBRAIC EFFECTS WITH EXISTENTIAL TYPES

In this section we introduce existential types for abstract effects. We motivate the existential types for abstract effects with an example of an abstract variable type. Then we present a use case of the extended system, which is modeling the path-dependent effects presented by [Melicher et al. 2020].

4.1 Motivating Example

Consider a type `Var` that provides two public functions, `get` and `set`, that are annotated with effects, but the concrete implementation that causes effects is not revealed to the client. This construction ensures that any program that reads or writes to a value of type `Var` is annotated with the correct

effect. Therefore we can see that, in a restrictive effect system, it is useful to be able to hide the implementation of a computational effect.

In this section, we will show that this form of implementation hiding is also important for building modular software with algebraic effects and handlers through an example of mutable state. Mutable state as an algebraic effect is often represented by a `state` effect with two operations `get` and `set`. In this example, we assume that our mutable state can store or access an integer.

```
operation get : Unit -> Int
operation set : Int -> Unit
```

The handler of a state effect is usually defined in the following way:

```
handler hstate = {
| return x -> λ_:Int. return x
| get(_; k) -> λs:Int. (k s) s
| set(s; k) -> λ_:Int. (k ()) s
}
```

This handler would transform the handled computation into a lambda expression that receives a state as an argument. In the return clause, the argument is ignored. In the clause handling `get`, the state argument is passed into the continuation k. Since the continuation $k\ s$ is already transformed into a function that expects a state, we pass s to the computation $k\ s$. The clause for the `set` operation is similar except that we ignore the state argument, but pass the argument obtained from the `set` operation.

Now we consider a module that provide access to a single variable, where read and write to the variable cannot bypass the effect checking because the implementation details are hidden by the interface. it is natural to extend our calculus with record type and implement the module as follows:

```
val var =
  < read ↪ λx:Unit. get(); y.return y),
    write ↪ λx:Int. set(x; y.return ()),
    handler ↪ λc: Unit → {state} Int. (with hstate handle c ()) 0 >
```

The module provides functions `read`, `write`, and a handler function `handler` that determines the semantics for operations in functions `read` and `write`. However, this encoding of the module `var` does not enforce that the operations `get` and `set` are always handled by the `handler` function. In fact, any client that calls function `read` and `write` can write its own handler to handle the effects. So in order to solve this issue, we introduce existential type to define the module:

```
val var =
  pack
  < {get, set},
    < read ↪ λx:Unit. get(); y.return y),
      write ↪ λx:Int. set(x; y.return ()),
      handler ↪ λc: Unit → {state} Int. (with hstate handle c ()) 0 >
  > as ∃state. ((read : Unit -> {state} Int)
              ×(write : Int -> {state} Unit)
              ×(handler : Unit -> {state} Int -> {} Int))
```

This encoding of module defines an abstract effect `state` on top of the effects `get` and `set`, and export the module as an existential type that hides the definition of effect `state`, therefore enforcing that the client of this module can only handle effect `state` by calling the `handler` function.

The embedding design presented in section 3.1 helps to ensure that the abstraction does not break. Imagine a client of module `var` that calls `read` and handles it by calling `handler`.

```
open var as (state, v) in (v.handler v.read)
```

In this example, the `state` effect in function `read` is correctly handled by the handler `handle`, and the result of the computation is 0. Note that the abstraction barrier is still preserved if some handler attempts to handle the abstract effect. For example, let handler `hget` be a handler that handles effect `get`. And the client code uses `hget` to handle the effect in method `read`.

```
handler hget = {
  return x -> x
  get (x; k) -> 1
}

open var as (state, v) in
  (v.handler (λ x: Unit. with hget handle v.read ()))
```

Assume that the open expression is an i-expression. When opening the module `var`, we assign the opened expression `v` to a new agent j that knows the definition of the `state` effect. Because the `hget` handler is evaluating in the agent i, it would not be able to handle the effect `state` of the function `v.read`. Instead, the `state` effect will be handled by the handler provided by the module `var`, therefore ensuring that the abstraction information is not leaked.

4.2 Encoding Abstract Effects Using Algebraic Effects

Our discussion of abstraction of algebraic effects has been focusing exclusively on purely functional programming. However, as shown by Melicher et al. [2020], the expressiveness provided by abstract effects enables programmers to develop secure programs when side-effects are in play. Specifically, abstract effects are great for controlling the usage of sensitive system resources such as file system and network. In this section, we show that our effect system provides a foundation for expressing abstract effect with security applications.

Melicher et al. [2020] and presented a design of object-oriented effect system which leverages effect members to support effect abstraction: the ability to define higher-level effects in terms of lower-level effects, to hide that definition from clients of an abstraction, and to reveal partial information about an abstract effect through effect bounds. In this paper we no longer use the object-oriented formalization but instead extend the agent-based lambda calculus with the existential types as a foundation to support effect abstraction. Now we will look at different aspects of the original abstract effect system and discuss how we can incorporate them in the new setting with algebraic effects.

4.2.1 Running Example. We begin by encoding the running example presented in Melicher et al. [2020] which demonstrates the key feature of abstract effects. The original example shows a type and a module implementing the logging facility in the text-editor application and is shown in figure 8.

Consider the code in Fig. 8 that shows a type and a module implementing the logging facility of the text-editor application. In the given implementation of the `Logger` type, the `logger` module accesses the log file. All modules of type `Logger` must have two methods: the `readLog` method that returns the content of the log file and the `updateLog` method that appends new entries to the log file. In addition, the `Logger` type declares two *abstract* effects, `ReadLog` and `UpdateLog`, that are produced by the corresponding methods. These effects are abstract because they are not given a definition in the `Logger` type, and so it is up to the module implementing the `Logger` type to define what they mean. The effect names are user-defined, allowing the choice of meaningful names.

The `logger` module implements the `Logger` type. To access the file system, an object of type `File` (shown in Fig. 8) is passed into `logger` as a parameter. The `logger` module's effect declarations are those of the `Logger` type, except now they are *concrete*, i.e., they have specific definitions. The `ReadLog`

```
resource type Logger
  effect ReadLog
  effect UpdateLog
  effect readLog(): {this.ReadLog} String
  effect updateLog(newEntry: String): {this.UpdateLog} Unit

module def logger(f: File): Logger
  effect ReadLog = {f.Read}
  effect UpdateLog = {f.Append}
  def readLog(): {ReadLog} String = f.read()
  def updateLog(newEntry: String): {UpdateLog} Unit = f.append(newEntry)

resource type File
  effect Read
  effect Write
  effect Append
  ...
  def read(): {this.Read} String
  def write(s: String): {this.Write} Unit
  def append(s: String): {this.Append} Unit
  ...
```

Fig. 8. The logging facility in the text-editor application

effect of the `logger` module is defined to be the `Read` effect of the `File` object, and accordingly, the `readLog` method, which produces the `ReadLog` effect, calls `f`'s `read` method. Similarly, the `UpdateLog` effect of the `logger` module is defined to be `f.Append`, and accordingly, the `updateLog` method, which produces the `UpdateLog` effect, calls `f`'s `append` method. In general, effects in a module or object definition must always be concrete, whereas effects in a type definition may be either abstract or concrete.

Using the existential type, the type `Logger` can be translated to the following type, note that we only translate one abstract effect `ReadLog` and one method `readLog` to make the code more readable. We assume that the type String is built into the language.

```
type Logger = ∃ReadLog.⟨readLog : Unit → {ReadLog} String⟩
```

Similarly, the `File` type can be implemented as follows. Again we leave only one abstract effect and one member function to maintain simplicity of the example.

```
type File = ∃Read.⟨read : Unit → {Read} String⟩
```

Then we are finally able to encode the functor `logger`, which receives a value of type `File` and returns a `Logger`.

```
logger : File → {} Logger =
  λf:File. open f as (fRead, fBody) in
  pack (fRead, ⟨readLog ↪ λx: Unit. fBody.read ()⟩) as
    ∃ReadLog.⟨readLog : Unit → {ReadLog} String⟩
```

4.2.2 Effect Abstraction. Effect abstraction is the ability to define higher-level effects in terms of lower-level effects and potentially to hide that definition from clients of an abstraction. In the `logger` module above, we lifted the low-level file resource into a higher-level logging facility, and defined higher-level effects `ReadLog` as an abstraction of the lower-level effect `Read`. So application code can reason in terms of effects of higher-level resources when appropriate.

4.2.3 Effect Aggregation. The authors of [Melicher et al. 2020] argued that effect aggregation can make the effect system less verbose. The original example declares an effect `UpdateLog` as the sum of two effects `f.Read` and `f.Write`.

```
module def logger(f: File): Logger
effect UpdateLog = {f.Read, f.Write}
def updateLog(newEntry: String): {this.UpdateLog} Unit
...
```

Although our new calculus is inherently more verbose than the language based on path-dependent effects, it can still encode effect aggregation. First we let the `File` type contains two abstract effects by existential quantification:

```
type File = ∃Read. ∃Write. ...
```

Then we can define logger functor. We first open the module `f`, then define an existential package where the abstract effect is defined as a sum of the two effects from the module `f`:

```
logger : File → {} Logger =
  λ f : File. open f as (fRead, x) in open x as (fWrite, y) in
  pack ({fRead, fWrite}, ⟨updateLog↪ ...⟩) as
    (∃UpdateLog. ⟨updateLog: String → {UpdateLog} Unit⟩)
```

Again the functor `logger` receives a `File` and returns a `Logger`. The functor opens the `f` module twice to get the two effect labels `fRead` and `fWrite` that are exported by `f`. Then the `logger` returns an existential package that hides the two effect as an abstract effect `UpdateLog`. This design achieves effect aggregation by combining the two lower-level effects into one higher-level effect.

5 FORMALIZATION OF EXISTENTIAL TYPES

In the previous sections, we have introduced the core calculus of abstract algebraic effects via embeddings. However it is impractical for requiring programmers to explicitly annotate each program component with embeddings. So we present a top level language where the annotations are implicitly added during the evaluation of the program.

According to Mitchell and Plotkin [1988], there is a correspondence between abstract data types and existential types. Existential types are often used as a foundation for expressing type abstraction in module systems. The calculus introduced in this section contains a form of existential type that provides abstraction mechanisms for algebraic effects. The values that have existential type would generate new agents during the evaluation of the program and automatically separate program components with different knowledge on effect abstraction, so programmers would not need to explicitly work with agents and embeddings.

5.1 Syntax

Most of the syntax remains the same for our new language. The existential type $\exists f.\ \tau$ is added as an expression type. The intuition of the type is that the value of this type is a value of type $\{\varepsilon/f\}\tau$ for some effect type ε.

There are two new forms of expressions: The introduction form of the existential type, $\mathsf{pack}\ (\varepsilon, e)\ \mathsf{as}\ \exists f.\ \tau$ and the elimination form, $\mathsf{open}\ e\ \mathsf{as}\ (f, x)\ \mathsf{in}\ e'$. The `pack` expression creates existential package that contains an effect type ε and an expression e. The open expression opens up an existential package and substitutes the expression in the package for the variable x.

We only introduce one form of value: the existential package $\mathsf{pack}\ (\varepsilon, v)\ \mathsf{as}\ \exists f.\ \tau$, where the packed expression is already evaluated to a value.

$$
\begin{array}{lll}
\text{(agents)} & & i, j ::= \{1 \ldots n\} \\
\text{(lists)} & & l ::= i \mid il \\
\text{(expression types)} & & \tau ::= 1 \mid \tau \to \sigma \mid \exists f.\, \tau \\
\text{(computation types)} & & \sigma ::= \{\varepsilon\}\tau \\
\text{(effect types)} & & \varepsilon ::= \cdot \mid f, \varepsilon \mid op, \varepsilon \\
\text{(i-values)} & & v_i ::= ()_i \mid \lambda x_i : \tau.\, c_i \mid \ \mathsf{pack}\ (\varepsilon, v)\ \mathsf{as}\ \exists f.\, \tau \\
\text{(i-expressions)} & & e_i ::= x_i \mid v_i \mid [e_j]^{\tau}_{l} \mid \ \mathsf{pack}\ (\varepsilon, e)\ \mathsf{as}\ \exists f.\, \tau \mid \ \mathsf{open}\ e\ \mathsf{as}\ (f, x)\ \mathtt{in}\ e' \\
\text{(i-computations)} & & c_i ::= \mathtt{return}\ e_i \mid op(e_i, y.c_i) \mid \mathsf{do}\ x \leftarrow c_i\ \mathtt{in}\ c_i' \mid e_i\ e_i' \\
& & \quad \mid \mathtt{with}\ h_i\ \mathsf{handle}\ c_i \mid \ [c_j]^{\sigma}_{l} \mid [op]^{\varepsilon}_{l}(e_i, y_i.c_i) \\
\text{(i-handler)} & & h_i ::= \mathtt{handler}\ \{\mathtt{return}\ x_i \mapsto c_i^r, op^1(x_i^1, k^1) \mapsto c_i^1 \ldots op^n(x_i^n, k^n) \mapsto c_i^n\}
\end{array}
$$

Fig. 9. Syntax for Existential Effects

5.2 Dynamic Semantics

$\boxed{\langle\{\Delta\}, e\rangle \mapsto \langle\{\Delta\}, e\rangle}$

$$\frac{\langle\{\Delta\}, e\rangle \mapsto \langle\{\Delta'\}, e'\rangle}{\langle\{\Delta\},\ \mathsf{pack}\ (\varepsilon, e)\ \mathsf{as}\ \exists f.\, \tau\rangle \mapsto \langle\{\Delta'\},\ \mathsf{pack}\ (\varepsilon, e')\ \mathsf{as}\ \exists f.\, \tau\rangle}\ \text{(E-PACK)}$$

$$\frac{f\ \mathsf{fresh} \quad j\ \mathsf{fresh} \quad \Delta'_j = \Delta_i[f \mapsto \varepsilon] \quad \forall \Delta_k \in \{\Delta\}, \Delta'_k = \Delta_k[f \mapsto f]}{\langle\{\Delta\},\ \mathsf{open}\ (\ \mathsf{pack}\ (\varepsilon, e)\ \mathsf{as}\ \exists f.\, \tau)\ \mathsf{as}\ (f, x)\ \mathtt{in}\ e'\rangle \mapsto \langle\{\Delta'\}, \{[e]^{\tau}_{j}/x\}e'\rangle}\ \text{(E-OPEN1)}$$

$$\frac{\langle\{\Delta\}, e\rangle \mapsto \langle\{\Delta'\}, e'\rangle}{\langle\{\Delta\},\ \mathsf{open}\ e\ \mathsf{as}\ (f, x)\ \mathtt{in}\ e''\rangle \mapsto \langle\{\Delta'\},\ \mathsf{open}\ e'\ \mathsf{as}\ (f, x)\ \mathtt{in}\ e''\rangle}\ \text{(E-OPEN2)}$$

$$\frac{}{\langle\{\Delta\}, [\ \mathsf{pack}\ (\varepsilon, v)\ \mathsf{as}\ \exists f.\, \tau]^{\exists f.\, \tau'}_{j}\rangle \mapsto \langle\{\Delta\},\ \mathsf{pack}\ (\varepsilon, [v]^{\{\varepsilon/f\}\tau'}_{j})\ \mathsf{as}\ \exists f.\, \tau'\rangle}\ \text{(E-EMBEDPACK)}$$

Fig. 10. Additional Dynamic Semantics for Existential Type

The semantics for existential type $\exists f.\, \tau$ hides the definition of the effect label f from the client of the value of this type. We leverage our previous design of multi-agent calculus to achieve information hiding. However, the previous design assumes that the type information for each agents is predetermined and does not change during evaluation. Since existential types generate new abstraction boundaries, we need different semantics that allow agents to be created during evaluation. Therefore, we modify the reduction rule to evaluate a pair that contains both the expression to evaluate and a context of type information. The idea to keep track of type information while evaluating terms was used by Grossman et al. [2000] to encode parametric polymorphism in their system.

In figure 10, we show the dynamics semantics for new constructs such as exists and open. In the rules we use the syntax $\Delta_i[f \mapsto \varepsilon]$ to express extending the type map Δ_i with a new projection from label $\mathtt{f}$ to effect ε.

We introduce the notation $\{\Delta\}$ to express a list of type maps for all agents in the context $\{\Delta_1, \ldots \Delta_n\}$. The type information of each agent can change, and new agents can be generated, the evaluation judgment now has the form $\langle\{\Delta\}, e\rangle \mapsto \langle\{\Delta\}, e\rangle$.

$$\boxed{\{\Delta\} \mid \Gamma \vdash e_i : \tau}$$

$$\frac{\{\Delta\} \mid \Gamma \vdash e : \{\varepsilon/f\}\tau}{\{\Delta\} \mid \Gamma \vdash \text{ pack } (\varepsilon, e) \text{ as } \exists f.\ \tau : \exists f.\ \tau} \text{ (T-PACK)}$$

$$\frac{\{\Delta\} \mid \Gamma \vdash e : \exists f.\ \tau \quad \forall \Delta_i \in \{\Delta\}, \Delta_i' = \Delta_i[f \mapsto f] \quad \{\Delta'\} \mid \Gamma, x : \tau \vdash e' : \tau'}{\{\Delta\} \mid \Gamma \vdash \text{ open } e \text{ as } (f, x) \text{ in } e' : \tau'} \text{ (T-OPEN)}$$

Fig. 11. Additional Static Semantics for Existential Effects

(E-Pack) shows the congruence rule for reduction of a pack expression. (E-Open) opens an existential package: This rule requires f to be a fresh label, which achievable by applying alpha conversion in τ. j is a fresh agent. A new type map for agent j extends the type map for i by mapping f to ε. Every existing type map in $\{\Delta\}$ is extended by mapping f to itself. Finally, $[e]_j^\tau$ is substituted for x in e'.

(E-EmbedPack) shows how the embedding interacts with existential packages. The existential package is lifted out of the embedding, and the value v becomes an embedded j-value with type annotation $\{\varepsilon/f\}\tau'$.

The reduction rules for remaining terms are not changed by the introduction of $\{\Delta\}$ and are therefore not shown.

5.3 Static Semantics

The typing rules also require the set of type maps, so the judgments have the form $\{\Delta\} \mid \Gamma \vdash e : \tau$ (T-Pack) assigns the type $\exists f.\ \tau$ to the existential package if the expression e has type $\{\varepsilon/f\}\tau$. The rule (T-Open) assigns the type τ' to the open expression if e has the existential type $\exists f.\ \tau$ and e' has type τ' given that the context is extended with variable x, and the set of type maps is extended with the effect label f.

Similar to the previous system, type relations ensure the correctness of the type annotation on embeddings. Since we introduced the existential type, we need to add an additional rule to the type relations:

$$\frac{\tau \leq_l \tau'}{\exists f.\ \tau \leq_l \exists f.\ \tau'} \ (R - Exists)$$

5.4 Type Safety

Lemma 5. *(Preservation for expression)*
If $\{\Delta\} \mid \Gamma \vdash e : \tau$ *and* $\langle\{\Delta\}, e\rangle \mapsto \langle\{\Delta'\}, e'\rangle$, *then* $\{\Delta'\} \mid \Gamma \vdash e' : \tau$

Lemma 6. *(Progress)*
If $\{\Delta\} \mid \Gamma \vdash e : \tau$ *then either* e *is a value or there exists* e' *and* $\{\Delta'\}$ *such that* $\langle\{\Delta\}, e\rangle \mapsto \langle\{\Delta'\}, e'\rangle$

6 DISCUSSION

6.1 Parametric Polymorphism

We introduced polymorphic effects in chapter **??** as a part of our restrictive effect system. However, we did not include polymorphism in our agent-based core calculus. Parametric polymorphism on effects would significantly increase the expressiveness of the language. For example, in the example of the `var` module presented in the previous section, the type of the argument to the `handle` function is `Unit -> {state} Int`. So it restricts the handled computation to only have the `state` effect, and is therefore unrealistic as we may want to handle computations with other effects as well.

```
val var =
  pack
  <{get, set},
    < read ↪ λx:Unit. get(); y.return y),
      write ↪ λx:Int. set(x; y.return ()),
      handle ↪ λc: Unit → {state} Int. (with hstate handle c ()) 0 >
  > as ∃state. (read : Unit -> {state} Int) *
               (write : Int -> {state} Unit) *
               (handle : Unit -> {state} Int -> {} Int)
```

Therefore it would be desirable to add universal quantifications on effect variables to the system. However, unlike existential quantification, which is a straightforward extension to our calculus, the universal effect introduces a problem that breaks the abstraction barrier. The following example illustrates the problem brought by parametric polymorphism on effects:

```
type B
  effect E {
   def op() : Unit
  }

module b : B
  ...

type A
  effect E
  def handle[F](c: Unit -> {F, this.E} Unit): {F} Unit
  def m(): {this.E} Unit

module a: A
  effect E = {b.E}
  def handle[F](c: Unit -> {F, this.E} Unit): {F} Unit
     handle
       c()
     with
       b.op() -> resume ()
  def m(): {this.E} Unit
     b.op()
```

The module `a` defines a polymorphic handle function that handles a computation with effect `a.E` and a polymorphic effect `F`. Since effect `a.E` is defined abstract in the type `A`, the client of this module should not observe the fact that effect `a.E` is equivalent to `b.E`. However, the `client1` in the following code passes effect `b.E` as the polymorphic effect into the `handle` function, and because the implementation of `handle` function handles the effect `b.E`, the operation `b.op` would be handled by `a.handle`. So client1 be would be surprised by that the effect `b.E` is handled, and the desired output is not printed. In comparison, the client2 code passes an unrelated effect `c.E` to the handling function and observes the line “desired output" is printed. This example illustrates that the clients can actually observe the implementation of the effect `a.E`, which is supposed to be opaque.

```
//Client1: Prints nothing
handle
  a.handle[b.E] (
     () => b.op(); a.m()
  )
```

```
6  with
7      b.op() -> print "desired output"
8
9  //Client2: Prints "desired output"
10 handle
11   a.handle[c.E] (
12     () => c.op(); a.m()
13   )
14 with
15     c.op() -> print "desired output"
```

6.2 Effect Bounds

Effect bounds are a useful tool for making the effect system more expressive. Currently, our calculus does not support bounded quantification because its formalization is different from the path-dependent formalization we previously developed to express effect bounds.

One possible direction to achieve this is through the use of bounded existential types. Because we use existential types to express information hiding on effect types, it is natural to adopt the technique of bounded existential types to achieve the idea of effect bounds. It could be interesting to explore how subeffecting introduced by effect bounds interacts with the mechanism of agent-based type information.

7 RELATED WORK

The issue of abstract algebraic effects was originally raised by Leijen [2018], but was not discussed in depth. Biernacki et al. [2019] introduced a core calculus called λ^{HEL} with abstract algebraic effects. However, there are multiple distinctions that distinguish our effect system from λ^{HEL}. First, we adopted the agent-based syntax that syntactically distinguishes each module by assigning them to different agents. This design allows us to reason about the code using the information provided by agents, in comparison, the coercion-based solution used by λ^{HEL} does not provide information on modules. Moreover, because our calculus is more related to the top-level modules, our calculus can be simply extended with existential types, which serves as an abstraction to represent module systems for a high-level language. The benefit of this design is that the programmer would not need to write embeddings explicitly, as embeddings are generated as a semantic object when a value of existential type is evaluated. In comparison, it is unclear from the paper [Biernacki et al. 2019] how a top-level language with a module system could be translated to the coercion-based calculus λ^{HEL}.

Melicher et al. [2020] present an object-oriented effect system that supports effect abstraction by leveraging the mechanism of effect abstraction, and argue that abstract effects are useful for security applications. Their system provides an expressive mechanism for expressing effect abstractions, but effects in their systems do not carry semantic meanings and cannot be handled by handlers, which makes their system unsuitable for modeling various computational effects such exceptions and other control-flow abstractions. In contrast, our system not only achieves similar level expressiveness power with effect abstractions, but is also able to express more language constructs with the help of effect handlers.

Zhang and Myers [2019] describe a design of algebraic effects that preserves abstraction in the setting of parametric functions. If a function does not statically know about an algebraic effect, that effect tunnels through that function. This is different from our form of abstraction, in which the definition of an effect is hidden from clients.

The Effekt library by Brachthauser et al. [2020] explores algebraic effects as a library of the Scala programming language. In their paper, several forms of effect compositions are discusses, which are similar to one of the use case of our effect system discussed in section 4.2. However, their system does not provide an effect abstraction mechanism similar to ours.

8 CONCLUSION

Effect systems have been actively studied for nearly four decades, but they are not widely used in the software development process because little attention is paid to improve the usability of effect systems when developing large and complex software. On the other hand, type abstraction is an invaluable tool to software designers, which enables programmers to reason about different components of programs. Therefore, we explored a different way to achieve effect type abstraction within existing frameworks of algebraic effects and handlers.

We have presented a core calculus for abstract algebraic effects, which ensures the correctness of abstraction by using embeddings to keep track of the type information during evaluation. We have provided proofs of the type soundness of our calculus. Moreover, we have added existential types for effects to our system as a foundation for a module system with abstract effect types. And we have discussed a potential use case for abstract algebraic effect by presenting an implementation of a secure logging facility that leverages abstract algebraic effects.

REFERENCES

Dariusz Biernacki, Maciej Piròg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting Algebraic Effects. In *Symposium on Principles of Programming Languages*.

JONATHAN IMMANUEL Brachthauser, PHILIPP SCHUSTER, and KLAUS OSTERMANN. 2020. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *Journal of Functional Programming* 30 (2020), e8. https://doi.org/10.1017/S0956796820000027

Oliver Bračevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. 2018. Versatile Event Correlation with Algebraic Effects. *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 67 (2018), 31 pages. https://doi.org/10.1145/3236762

Dan Grossman, Greg Morrisett, and Steve Zdancewic. 2000. Syntactic Type Abstraction. *ACM Trans. Program. Lang. Syst.* 22, 6 (Nov. 2000), 1037–1080. https://doi.org/10.1145/371880.371887

Joseph R. Kiniry. 2006. *Advanced Topics in Exception Handling Techniques.* Springer-Verlag, Chapter Exceptions in Java and Eiffel: Two Extremes in Exception Design and Application. http://dl.acm.org/citation.cfm?id=2124243.2124264

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types, In Mathematically Structured Functional Programming. https://www.microsoft.com/en-us/research/publication/koka-programming-with-row-polymorphic-effect-types-2/

Daan Leijen. 2018. *Algebraic Effect Handlers with Resources and Deep Finalization.* Technical Report MSR-TR-2018-10. 35 pages. https://www.microsoft.com/en-us/research/publication/algebraic-effect-handlers-resources-deep-finalization/

Paul Levy, John Power, and Hayo Thielecke. 2003. Modelling Environments in Call-By-Value Programming Languages. *Information and Computation* 185 (08 2003). https://doi.org/10.1016/S0890-5401(03)00088-9

Yuheng Long, Yu David Liu, and Hridesh Rajan. 2015. Intensional Effect Polymorphism. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs)*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 346–370. https://doi.org/10.4230/LIPIcs.ECOOP.2015.346

John M. Lucassen. 1987. *Types and Effects towards the Integration of Functional and Imperative Programming.* Ph.D. Dissertation. Massachusetts Institute of Technology.

John M. Lucassen and David K. Gifford. 1988. Polymorphic Effect Systems. In *Symposium on Principles of Programming Languages* (San Diego, California, USA). 11. https://doi.org/10.1145/73560.73564

Darya Melicher, Anlun Xu, Jonathan Aldrich, Alex Potanin, and Zhao Valerie. 2020. Bounded Abstract Effects: Applications to Security. (2020).

John C. Mitchell and Gordon D. Plotkin. 1988. Abstract Types Have Existential Type. *ACM Trans. Program. Lang. Syst.* 10, 3 (July 1988), 470–502. https://doi.org/10.1145/44501.45065

Gordon Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11 (02 2003), 69–94. https://doi.org/10.1023/A:1023064908962

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems.*

Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight Polymorphic Effects. In *European Conference on Object-Oriented Programming* (Beijing, China). 25. https://doi.org/10.1007/978-3-642-31057-7_13

Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. 2019. Monad Transformers and Modular Algebraic Effects: What Binds Them Together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell* (Berlin, Germany) *(Haskell 2019)*. Association for Computing Machinery, New York, NY, USA, 98–113. https://doi.org/10.1145/3331545.3342595

Valery Trifonov and Zhong Shao. 1999. Safe and Principled Language Interoperation. In *European Symposium on Programming Languages and Systems*.

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe Effect Handlers via Tunneling. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 5 (2019), 29 pages. https://doi.org/10.1145/3290318

A TYPE SAFETY THEOREMS FOR ALGEBRAIC EFFECTS AND HANDLERS

A.1 Lemmas

Lemma 7. *(Substitution)*
If $\Gamma, x_j : \tau' \vdash c_i : \sigma$ *and* $\Gamma \vdash e_j : \tau'$, *then* $\Gamma \vdash \{e_j/x_j\}c_i : \sigma$, *and*
If $\Gamma, x_j : \tau' \vdash e_i : \tau$ *and* $\Gamma \vdash e_j : \tau'$, *then* $\Gamma \vdash \{e_j/x_j\}e_i : \tau$

Proof. By rule induction on $\Gamma \vdash e : \tau$ and $\Gamma \vdash c : \sigma$

(T-Unit) Trivial

(T-Var) Trivial

(T-Lam)

$$\frac{\Gamma, x_j : \tau', x_i : \tau \vdash c_i : \sigma}{\Gamma, x_j : \tau' \vdash (\lambda x_i : \tau.\ c_i) : \tau \to \sigma}\ (\text{T-LAM})$$

By IH, we have $\Gamma, x_i : \tau \vdash \{e_j/x_j\}c_i : \sigma$
Then by (T-Lam) we have $\Gamma \vdash (\lambda x_i : \tau.\ \{e_j/x_j\}c_i) : \sigma$.
Which is equivalent to $\Gamma \vdash \{e_j/x_j\}(\lambda x_i : \tau.\ c_i) : \sigma$.

(T-EmbedExp) By inversion and IH

(T-Ret) Follows by induction hypothesis

(T-Op)

$$\frac{\Sigma(op) = \tau_A \to \tau_B \quad \Gamma \vdash e_i : \tau_A \quad \Gamma.y_i : \tau_B \vdash c_i : \{\varepsilon\}\tau \quad op \in \Delta_i(\varepsilon)}{\Gamma \vdash op(e_i; y_i.c_i) : \{\varepsilon\}\tau}\ (\text{T-OP})$$

By inversion we have $\Gamma, x_j : \tau' \vdash e_i : \tau_A$ and $\Gamma, x_j : \tau', y_i : \tau_B \vdash c_i : \{\varepsilon\}\tau$.
Since we can make y_i a fresh variable, we have $\Gamma, y_i : \tau_B, x_j : \tau' \vdash c_i : \{\varepsilon\}\tau$.
Then by IH we have $\Gamma \vdash \{e_j/x_j\}e_i : \tau_A$ and $\Gamma, y_i : \tau_B \vdash \{e_j/x_j\}c_i : \{\varepsilon\}\tau$.
By (T-Op) we have $\Gamma \vdash op(\{e_j/x_j\}e_i; y_i.\{e_j/x_j\}c_i) : \{\varepsilon\}\tau$
Therefore we have $\Gamma \vdash \{e_j/x_j\}(op(e_i; y_i.c_i)) : \{\varepsilon\}\tau$

(T-Seq)

$$\frac{\Gamma, x_j : \tau'' \vdash c_i : \{\varepsilon\}\tau \quad \Gamma, x_j : \tau'', x_i : \tau \vdash c_i' : \{\varepsilon\}\tau'}{\Gamma, x_j : \tau'' \vdash \ \mathtt{do}\ x_i \leftarrow c_i\ \mathtt{in}\ c_i' : \{\varepsilon\}\tau'}\ (\text{T-SEQ})$$

By IH, we have $\Gamma \vdash \{e_j/x_j\}c_i : \{\varepsilon\}\tau$
Since we can choose x_i as a fresh variable, we have $\Gamma, x_i : \tau, x_j : \tau'' \vdash c_i' : \{\varepsilon\}\tau'$
Then by IH we have $\Gamma, x_i : \tau \vdash \{e_j/x_j\}c_i' : \{\varepsilon\}\tau'$
Then the result follows by (T-Seq)

(T-App) Follows directly by applying IH.

(T-Handle)

$$\frac{\begin{array}{c} h_i = \{\ \mathtt{return}\ x \mapsto c^r, op^1(x;k) \mapsto c^1, \ldots, op^n(x;k) \mapsto c^n\} \\ \Gamma, x_j : \tau', x : \tau_A \vdash c^r : \{\varepsilon'\}\tau_B \\ \left\{\Sigma(op^i) = \tau_i \to \tau_i' \quad \Gamma, x_j : \tau', x : \tau_i, k : \tau_i' \to \{\varepsilon'\}\tau_B \vdash c^i : \{\varepsilon'\}\tau_B\right\}_{1 \le i \le n} \\ \Gamma, x_j : \tau' \vdash c_i : \{\varepsilon\}\tau_A \quad \varepsilon \setminus \{op^i\}_{1 \le i \le n} \subseteq \varepsilon' \end{array}}{\Gamma, x_j : \tau' \vdash \ \mathtt{with}\ h_i\ \mathtt{handle}\ c_i : \{\varepsilon'\}\tau_B}\ (\text{T-HANDLE})$$

Then handling clauses bind variables x and k in the handling computation c^i, so we can make them fresh variables that do not appear in context Γ. Then we can apply IH to typing judgements in the premise.

(T-Embed) Follows by applying IH

(T-EmbedOp) The proof is similar to the case for (T-Op)

□

Lemma 8. *If* $\Gamma \vdash c_i : \{\varepsilon\}\tau$ *then* $\overline{\Delta_i}(\varepsilon) = \varepsilon$

Proof. By induction on derivation of $\Gamma \vdash c : \sigma$. (T-Ret) has a premise the ensures the lemma is correct. For other rules, the result is immediate by applying IH.

□

Lemma 9. *If* $\varepsilon \leq_l \varepsilon'$, *then* $\varepsilon \setminus op \leq_l \varepsilon' \setminus op$

Proof. By induction on $\varepsilon \leq_l \varepsilon$. The proof is straightforward. □

Lemma 10. *If* $op \leq_l \varepsilon$, *then* $op \leq_l \varepsilon \setminus op'$

Proof. By induction on the derivation of $\varepsilon \leq_l \varepsilon$. If (R-Eff1) is used, then the proof is straightforward because the subset relation on the premise still holds. If (R-Eff2) is used, by inversion on (R-Eff2), we have $op \leq_l \varepsilon'$ and $\varepsilon' \leq_{l'} \varepsilon$. By IH we have $op \leq_l \varepsilon' \setminus op'$. By lemma 9 we have $\varepsilon' \setminus op' \leq_{l'} \varepsilon \setminus op'$. Then the result follows by (R-Eff2) □

Lemma 11. *If* $\tau \leq_l \tau'$ *then* $\tau' \leq_{rev(l)} \tau$

Proof. By induction on the type relation rules. The proof consists of simple arguments that follow directly from IH. □

A.2 Preservation

A.2.1 Proof of lemma 1 (Preservation for expressions). For all agent i, If $\Gamma \vdash e_i : \tau$ and $e_i \mapsto e_i'$, then $\Gamma \vdash e_i' : \tau$.

Proof. By induction on derivation of $e_i \mapsto e_i'$

(E-Congruence) By inversion on the typing rule for embedded expressions, we have $\Gamma \vdash e_j : \tau'$. By IH, we have $\Gamma \vdash e_j' : \tau'$. Then we use (E-Contruence) to derive $\Gamma \vdash [e_j']_j^\tau : \tau$

(E-Unit) Follows immediately from (T-Unit)

(E-Lambda) By inversion on (T-Embed), we have $\Gamma \vdash \lambda x_j : \tau'.\ c_j : \tau' \longrightarrow \sigma'$, where $\tau' \longrightarrow \sigma' \leq_{ji} \tau \longrightarrow \sigma$.
By inversion on (R-Arrow), we have $\tau' \leq_{ji} \tau$ and $\sigma' \leq_{ji} \sigma$
By inversion on (T-Lambda), we have $\Gamma, x_j : \tau' \vdash c_j : \sigma'$. And since x_i is a fresh variable in c_j, we have $\Gamma, x_i : \tau, x_j : \tau' \vdash c_j : \sigma'$
By lemma 11, we have $\tau \leq_{ij} \tau'$, and therefore $\Gamma, x_i : \tau \vdash [x_i]_i^{\tau'} : \tau'$
Then we can use the substitution lemma to derive $\Gamma, x_i : \tau \vdash \{[x_i]_i^{\tau'}/x_j\}c_j : \sigma'$.
Then by (T-Embed), we have $\Gamma, x_i : \tau \vdash [\{[x_i]_i^{\tau'}/x_j\}c_j]_j^\sigma : \sigma$
Then the result follows by (T-Lambda).

□

A.2.2 Proof of lemma 2 (Preservation for computations). If $\Gamma \vdash c_i : \{\varepsilon\}\tau$ and $c_i \longrightarrow c_i'$, then $\Gamma \vdash c_i' : \{\varepsilon\}\tau$

Proof. (Sketch) By induction on the derivation that $c_i \longrightarrow c_i'$. We proceed by the cases on the last step of the derivation.

(1) E-Ret: By inversion, $\Gamma \vdash e_i : \tau$. By preservation of expressions and IH, we have $\Gamma \vdash e_i' : \tau$. Then we can use E-Ret to derive $\Gamma \vdash c_i' : \{\varepsilon\}\tau$
(2) E-Op: Follow immediately from inversion and IH
(3) E-EmbedOp1: Follow immediately from inversion and IH

(4) E-EmbedOp2:

$$\frac{\overline{\Delta_i}(\varepsilon) = \varepsilon''}{[op_j]_l^{\varepsilon}(v_i; y_i.c_i) \longrightarrow [op_j]_l^{\varepsilon''}(v_i; y_i.c_i)} \text{ (E-EmbedOp2)}$$

We have the typing rule as follows:

$$\frac{\Sigma(op) = \tau_A \to \tau_B \quad \Gamma \vdash e_i : \tau_A \quad \Gamma.y_i : \tau_B \vdash c_i : \{\varepsilon'\}\tau \quad \overline{\Delta_i}(\varepsilon) \subseteq \overline{\Delta_i}(\varepsilon') \quad \Gamma \vdash op \leq_{li} \varepsilon}{\Gamma \vdash [op]_l^{\varepsilon}(e_i; y_i.c_i) : \{\overline{\Delta_i}(\varepsilon')\}\tau}$$

Since $\overline{\Delta_i}(\varepsilon'') = \varepsilon''$ and $\varepsilon'' = \overline{\Delta_i}(\varepsilon)$, we have $\overline{\Delta_i}(\varepsilon'') \subseteq \overline{\Delta_i}(\varepsilon')$. Then we can use T-EmbedOp to derive $\Gamma \vdash [op]_l^{\varepsilon''}(e_i; y_i.c_i) : \{\overline{\Delta_i}(\varepsilon')\}\tau$

(5) E-EmbedOp3: We have the typing rule as follows:

$$\frac{\Sigma(op) = \tau_A \to \tau_B \quad \Gamma \vdash e_i : \tau_A \quad \Gamma.y_i : \tau_B \vdash c_i : \{\varepsilon'\}\tau \quad \overline{\Delta_i}(\varepsilon) \subseteq \overline{\Delta_i}(\varepsilon') \quad \Gamma \vdash op \leq_{li} \varepsilon}{\Gamma \vdash [op]_l^{\varepsilon}(e_i; y_i.c_i) : \{\overline{\Delta_i}(\varepsilon')\}\tau}$$

By E-EmbedOp3, $op \in \overline{\Delta_i}(\varepsilon)$. So $op \in \overline{\Delta_i}(\varepsilon')$. By inversion on the typing rule, we have $\Gamma, y_i : \tau_B \vdash c_i : \{\varepsilon'\}\tau$. By lemma 8, we have $\Gamma, y_i : \tau_B \vdash c_i : \{\overline{\Delta_i}(\varepsilon')\}\tau$. Then we can use T-Op to derive the designed result $\Gamma \vdash op(e_i; y_i.c_i) : \{\overline{\Delta_i}(\varepsilon')\}\tau$

(6) E-EmbedOp4: We have the typing rule as follows:

$$\frac{\Sigma(op) = \tau_A \to \tau_B \quad \Gamma \vdash e_i : \tau_A \quad \Gamma.y_i : \tau_B \vdash c_i : \{\varepsilon'\}\tau \quad \overline{\Delta_i}(\varepsilon) \subseteq \overline{\Delta_i}(\varepsilon') \quad \Gamma \vdash op \leq_{li} \varepsilon}{\Gamma \vdash [op]_l^{\varepsilon}(e_i; y_i.c_i) : \{\overline{\Delta_i}(\varepsilon')\}\tau}$$

By lemma 10, we have $op \leq_{li} \varepsilon \setminus op'$. By inversion on the typing rule, we have $\Gamma, y_i : \tau_B \vdash c_i : \{\varepsilon'\}\tau$ and $\varepsilon \subseteq \overline{\Delta_i}(\varepsilon')$. So $\varepsilon \setminus op' \subseteq \overline{\Delta_i}(\varepsilon')$. By lemma 8, we have $\Gamma, y_i : \tau_B \vdash c_i : \{\overline{\Delta_i}(\varepsilon')\}\tau$. Then we can apply T-EmbedOp again to derive $\Gamma \vdash [op]_l^{\varepsilon \setminus op'}(e_i; y_i.c_i) : \{\overline{\Delta_i}(\varepsilon')\}\tau$

(7) E-App1: Follows immediately by T-App

(8) E-App2: Follows immediately by T-App

(9) E-App3: By inversion of T-App, we $\Gamma \vdash (\lambda x_i : \tau.\ c_i) : \tau \to \sigma, \Gamma \vdash v_i : \tau$. By inversion of T-Lam, $\Gamma, x_i : \tau \vdash c_i : \sigma$. By substitution lemma, we have $\Gamma \vdash \{v_i/x_i\}c_i : \sigma$.

(10) E-Seq1: Follows immediately by T-Seq and IH.

(11) E-Seq2: By inversion on T-Seq, we have $\Gamma \vdash \texttt{return}\ v_i : \{\varepsilon\}\tau$ and $\Gamma, x_i : \tau \vdash c_i' : \{\varepsilon\}\tau'$. By inversion on T-Ret, we have $\Gamma \vdash v_i : \tau$. Then by substitution lemma we have $\Gamma \vdash \{v_i/x\}c_i' : \{\varepsilon\}\tau'$.

(12) E-Seq3:

$$\frac{}{\texttt{do}\ x \leftarrow op_i(v_i; y_i.c_i)\ \texttt{in}\ c_i' \longrightarrow op_i(v_i; y_i.\ \texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i')} \text{ (E-Seq3)}$$

By inversion of T-Seq, we have $\Gamma \vdash op_i(v_i; y_i.c_i) : \{\varepsilon\}\tau$ and $\Gamma, x : \tau \vdash c_i' : \{\varepsilon\}\tau'$. By inversion on T-OP, we have $\Gamma, y_i : \tau_B \vdash c_i : \{\varepsilon\}\tau$ and $op \in \varepsilon$ and $\Gamma \vdash v_i : \tau_A$. Then by T-Seq, we have $\Gamma, y_i : \tau_B \vdash \texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i' : \{\varepsilon\}\tau'$. Then we can use T-Op to derive $\Gamma \vdash op_i(v_i; y_i.\ \texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i') : \{\varepsilon\}\tau'$.

(13) E-Seq4

$$\frac{\Delta_i(\varepsilon) = \varepsilon \quad op \notin \varepsilon}{\texttt{do}\ x \leftarrow [op_j]_l^{\varepsilon}(v_i; y_i.c_i)\ \texttt{in}\ c_i' \longrightarrow [op_j]_l^{\varepsilon}(v_i; y_i.\ \texttt{do}\ x \leftarrow c_i\ \texttt{in}\ c_i')} \text{ (E-Seq4)}$$

$$\frac{\Gamma \vdash c_i : \{\varepsilon'\}\tau \quad \Gamma, x_i : \tau \vdash c_i' : \{\varepsilon'\}\tau'}{\Gamma \vdash \texttt{do}\ x_i \leftarrow c_i\ \texttt{in}\ c_i' : \{\varepsilon'\}\tau'} \text{ (T-seq)}$$

By inversion on T-Seq, we have $\Gamma \vdash [op_j]_l^{\varepsilon}(v_i; y_i.c_i) : \{\varepsilon'\}\tau$ and $\Gamma, x : \tau \vdash c_i' : \{\varepsilon'\}\tau'$. Then by inversion on T-EmbedOp, we have $\Gamma, y_i : \tau_B \vdash c_i : \{\varepsilon'\}\tau$, $\overline{\Delta_i}(\varepsilon) \subseteq \varepsilon'$. Then by T-Seq, we have $\Gamma, y_i : \tau_B \vdash \ \mathsf{do}\ x \leftarrow c_i\ \mathsf{in}\ c_i' : \{\varepsilon'\}\tau'$. Then by T-EmbedOp, we have $\Gamma \vdash [op]_l^{\varepsilon}(v_i; y_i.\ \mathsf{do}\ x \leftarrow c_i\ \mathsf{in}\ c_i') : \{\varepsilon'\}\tau'$

(14) E-Handle1: Follows immediately by inversion on T-Handle and IH

(15) E-Handle2: By T-Handle, we have $\Gamma \vdash \ \mathtt{with}\ h_i\ \mathtt{handle}\ \mathtt{return}\ v_i : \{\varepsilon'\}\tau_B$. By inversion on T-Handle, we have $\Gamma, x_i : \tau_A \vdash c_i' : \{\varepsilon\}\tau_B$, and $\Gamma \vdash \ \mathtt{return}\ v_i : \{\varepsilon\}\tau_A$. By inversion on T-Ret, we have $\Gamma \vdash v_i : \tau_A$. Then by substitution lemma, we have $\Gamma \vdash \{v_i/x_i\}c_i' : \{\varepsilon'\}\tau_B$.

(16) E-Handle3

$$\frac{op(x_i; k) \mapsto c_i' \in h_i \quad \Sigma(op) = \tau_i \to \tau_i'}{\mathtt{with}\ h_i\ \mathtt{handle}\ op(v; y_i.c_i) \longrightarrow \{v_i/x_i\}\{(\lambda y_i : \tau_i'.\ \mathtt{with}\ h_i\ \mathtt{handle}\ c_i)/k\}c_i'}$$

By T-Handle, we have $\Gamma \vdash \ \mathtt{with}\ h_i\ \mathtt{handle}\ op(v; y_i.c_i) : \{\varepsilon'\}\tau_B$. By inversion on T-Handle, we have $\Gamma, x_i : \tau_i, k : \tau_i' \to \{\varepsilon'\}\tau_B \vdash c_i' : \{\varepsilon'\}\tau_B$, and $\Gamma \vdash op(v; y_i.c_i) : \{\varepsilon\}\tau_A$. By inversion on T-Op, we have $\Gamma \vdash v_i : \tau_i$ and $\Gamma, y_i : \tau_i' \vdash c_i : \{\varepsilon\}\tau_A$. By T-Handle, we have $\Gamma, y_i : \tau_i' \vdash \mathtt{with}\ h_i\ \mathtt{handle}\ c_i : \{\varepsilon'\}\tau_B$. Then by T-Lam, we have $\Gamma \vdash \lambda y_i : \tau_i'.\ \ \mathtt{with}\ h_i\ \mathtt{handle}\ c_i : \tau_i' \to \{\varepsilon'\}\tau_B$. Then, by substitution lemma, we have $\Gamma \vdash \{v_i/x_i\}\{(\lambda y_i : \tau_i'.\ \mathtt{with}\ h_i\ \mathtt{handle}\ c_i)/k\}c_i' : \{\varepsilon'\}\tau_B$.

(17) E-Handle4:

$$\frac{\Delta_i(\varepsilon) = \varepsilon \quad op \notin \varepsilon}{\mathtt{with}\ h_i\ \mathtt{handle}\ [op]_l^{\varepsilon}(v_i, y_i.c_i) \longrightarrow [op]_l^{\varepsilon}(v_i; y_i.\ \mathtt{with}\ h_i\ \mathtt{handle}\ c_i))} \ \text{(E-Handle4)}$$

$$\frac{\begin{array}{c} h_i = \{\ \mathtt{return}\ x \mapsto c^r, op^1(x; k) \mapsto c^1, \ldots, op^n(x; k) \mapsto c^n\} \\ \Gamma, x : \tau_A \vdash c^r : \{\varepsilon'\}\tau_B \\ \left\{\Sigma(op^i) = \tau_i \to \tau_i' \quad \Gamma, x : \tau_i, k : \tau_i' \to \{\varepsilon'\}\tau_B \vdash c^i : \{\varepsilon'\}\tau_B\right\}_{1 \le i \le n} \\ \Gamma \vdash c_i : \{\varepsilon''\}\tau_A \quad \varepsilon'' \setminus \{op^i\}_{1 \le i \le n} \subseteq \varepsilon' \end{array}}{\Gamma \vdash \ \mathtt{with}\ h_i\ \mathtt{handle}\ c_i : \{\varepsilon'\}\tau_B} \ \text{(T-handle)}$$

By T-Handle, we have $\Gamma \vdash \ \mathtt{with}\ h_i\ \mathtt{handle}\ [op]_l^{\varepsilon}(v; y_i.c_i) : \{\varepsilon'\}\tau_B$. By inversion on T-Handle, we have $\Gamma \vdash [op]_l^{\varepsilon}(v; y_i.c_i) : \{\varepsilon''\}\tau_A$ and $\varepsilon'' \setminus \{op^i\} \subseteq \varepsilon'$. By inversion on T-EmbedOp, we have $\Gamma \vdash v_i : \tau_i, \Gamma, y_i : \tau_i' \vdash c_i : \{\varepsilon''\}\tau_A$ and $\varepsilon \subseteq \varepsilon''$. Since ε doesn't contain any concrete operation, we have $\varepsilon \subseteq \varepsilon'' \setminus \{op^i\} \subseteq \varepsilon'$. Then by T-Handle, we have $\Gamma, y_i : \tau_i' \vdash \ \mathtt{with}\ h_i\ \mathtt{handle}\ c_i : \{\varepsilon'\}\tau_B$. Then, we use T-EmbedOp to derive $\Gamma \vdash [op]_l^{\varepsilon}(v_i; y_i.\ \mathtt{with}\ h_i\ \mathtt{handle}\ c_i) : \{\varepsilon'\}\tau_B$

(18) E-Embed1: Follows immediately from Inversion and IH

(19) E-Embed2: By typing rule, we have $\Gamma \vdash [\ \mathtt{return}\ v_j]_l^{\{\varepsilon\}\tau} : \{\varepsilon\}\tau$. By inversion on the typing rule, we have $\Gamma \vdash \ \mathtt{return}\ v_j : \{\varepsilon'\}\tau'$ such that $\{\varepsilon\}\tau' \le_{li} \{\varepsilon\}\tau$. By inversion on R-Sigma, we have $\tau' \le_{li} \tau$. Then by T-EmbedExp, we have $\Gamma \vdash [v_j]_l^{\tau} : \tau$. Then by T-Ret, we have $\Gamma \vdash \ \mathtt{return}\ [v_j]_l^{\tau} : \{\varepsilon\}\tau$. $\Gamma \vdash [\ \mathtt{return}\ v_j]_l^{\{\varepsilon\}\tau} : \{\varepsilon\}\tau$

(20) E-Embed3:

$$\frac{\Sigma(op) = \tau_A \to \tau_B}{[op(v_j; y_j.c_j)]_l^{\{\varepsilon\}\tau} \longrightarrow [op]_l^{\varepsilon}([v_j]_j^{\tau_A}; y_i.\{[y_i]_i^{\tau_B}/y_j\}[c_j]_l^{\{\varepsilon\}\tau})} \ \text{(E-Embed3)}$$

By typing rule, we have $\Gamma \vdash op(v_j; y_j.c_j) : \{\varepsilon'\}\tau'$, where $\{\varepsilon'\}\tau' \le_{li} \{\varepsilon\}\tau$. By inversion on T-Op, we have $\Gamma \vdash v_j : \tau_A$, and $\Gamma, y_j : \tau_B \vdash c_j : \{\varepsilon'\}\tau'$. Then, by T-EmbedExp, we have $\Gamma \vdash [v_j]_j^{\tau_A} : \tau_A$. By substitution lemma, we have $\Gamma, y_i : \tau_B \vdash \{[y_i]_i^{\tau_B}/y_j\}c_j : \{\varepsilon'\}\tau'$. By

T-Embed, we have $\Gamma, y_i : \tau_B \vdash \{[y_i]_i^{\tau_B}/y_j\}[c_j]_l^{\{\varepsilon\}\tau} : \{\varepsilon\}\tau$. Then we can use T-EmbedOp to derive $\Gamma \vdash [op]_l^{\varepsilon}([v_j]_j^{\tau_A}; y_i.\{[y_i]_i^{\tau_B}/y_j\}[c_j]_l^{\{\varepsilon\}\tau}) : \{\varepsilon\}\tau$.

(21) E-Embed4:

$$\frac{\Sigma(op_k) = \tau_A \longrightarrow \tau_B \quad \Delta_j(\varepsilon') = \varepsilon' \quad op \notin \varepsilon'}{[[op_k]_{l'}^{\varepsilon'}(v_j; y_j.c_j)]_l^{\{\varepsilon\}\tau} \longrightarrow [op_k]_{l'jl}^{\varepsilon}([v_j]_j^{\tau_A}; y_i.\{[y_i]_i^{\tau_B}/y_j\}[c_j]_l^{\{\varepsilon\}\tau})} \text{ (E-Embed4)}$$

By typing rule, we have $\Gamma \vdash [op_K]_{l'}^{\varepsilon'}(v_j; y_j.c_j) : \{\varepsilon''\}\tau''$, where $\{\varepsilon''\}\tau'' \leq_{li} \{\varepsilon\}\tau$. By inversion on T-EmbedOp, we have $\Gamma \vdash v_j : \tau_A$ and $\Gamma, y_j : \tau_B \vdash c_j : \{\varepsilon''\}\tau''$. Then, by T-EmbedExp, we have $\Gamma \vdash [v_j]_j^{\tau_A} : \tau_A$. By substitution lemma, we have $\Gamma, y_i : \tau_B \vdash \{[y_i]_i^{\tau_B}/y_j\}c_j : \{\varepsilon''\}\tau''$. By T-Embed, we have $\Gamma, y_i : \tau_B \vdash \{[y_i]_i^{\tau_B}/y_j\}[c_j]_l^{\{\varepsilon\}\tau} : \{\varepsilon\}\tau$. Then we use T-EmbedOp to derive $\Gamma \vdash [op_k]_{l'jl}^{\varepsilon}([v_j]_j^{\tau_A}; y_i.\{[y_i]_i^{\tau_B}/y_j\}[c_j]_l^{\{\varepsilon\}\tau}) : \{\varepsilon\}\tau$.

□

A.3 Progress

A.3.1 Proof of lemma 3 (Progress for expressions). For agent i, if $\emptyset \vdash e_i : \tau$ then either $e_i = v_i$ or $e_i \longrightarrow e_i'$.

Proof. By induction on structure of e_i.

Case $e_i = ()$: e_i is already a value.

Case $e_i = \lambda x : \tau.c$: e_i is already a value.

Case $e_i = [e_j]_j^{\tau}$: By IH, either e_j is a j-value, or $e_j \longrightarrow e_j'$. If $e_j \longrightarrow e_j'$, then by (E-Congruence), $[e_j]_j^{\tau} \longrightarrow [e_j']_j^{\tau}$. If e_j is a value, then it is either () or $\lambda x_j : \tau'.c_j$. So e_i can be evaluated by (E-Unit) and (E-Lambda) correspondingly.

□

A.3.2 Proof of lemma 4 (Progress for computation). If $\emptyset \vdash c_i : \{\varepsilon\}\tau$ then either

(1) $c_i \longrightarrow c_i'$
(2) $c_i =$ `return` v_i
(3) $c_i = op(v_i; y_i.c_i')$
(4) $c_i = [op]_l^{\varepsilon}(v_i; y_i.c_i')$ and $op \notin \varepsilon$

Proof. By induction on structure of c_i.

Case $c_i =$ `return` e_i : Immediate by applying IH on e_i.

Case $c_i = op(e_i, y_i.c_i')$: Immediate by applying IH on e_i.

Case $c_i = [c_j]_j^{\sigma}$: By IH on c_j, c_j can either evaluates to another computation, or be a return statement, an operation call, or an embedded operation call. Then we can apply (E-Embed) rules to evaluate c_i accordingly.

Case $c_i = [op]_l^{\varepsilon}(e_i, y_i.c_i')$: Follows directly by applying IH on e_i.

Case $c_i = e_i\ e_i'$: If e_i or e_i' are not values, then (E-App1) or (E-App2) can be applied to c_i. Otherwise, (E-App3) could be applied.

Case $c_i =$ `do` $x \leftarrow c_i'$ `in` c_i'' : Follows directly from applying IH on c_i'.

Case $c_i =$ `with` h_1 `handle` c_i' : Follows directly from applying IH on c_i'.

□

B TYPE SOUNDNESS FOR EXISTENTIAL TYPES

B.1 Proof of theorem 5 (Preservation for expression)

Proof. By rule induction on the dynamic semantics of expressions.

(1) (E-Congruence): By inversion on typing judgement and applying IH.

(2) (E-Unit): By directly applying (T-Unit)

(3) (E-Lambda):

$$\frac{}{[\lambda x_j : \tau'.\ c_j]_j^{\tau\to\sigma} \longrightarrow \lambda x_i : \tau.\ [\{[x_i]_i^{\tau'}/x_j\}c_j]_{jl}^{\sigma}}\ (\text{E-Lambda})$$

By inversion on (T-EmbedExp), we have $\{\Delta\} \mid \Gamma \vdash [\lambda x_j : \tau'.\ c_j]_j^{\tau\to\sigma} : \tau \to \sigma$, and $\{\Delta\} \mid \Gamma \vdash \lambda x_j : \tau'.\ c_j : \tau' \to \sigma'$, where $\{\Delta\} \mid \Gamma \vdash \tau' \to \sigma' \leq_j \tau \to \sigma$. By inversion on (T-Lam), we have $\{\Delta\} \mid \Gamma, x_j : \tau' \vdash c_j : \sigma$. Then by substitution lemma, we have $\{\Delta\} \mid \Gamma, x_i : \tau \vdash \{[x_i]_i^{\tau'}/x_j\}c_j : \sigma'$. By (T-Embed), $\{\Delta\} \mid \Gamma, x_i : \tau \vdash [\{[x_i]_i^{\tau'}/x_j\}c_j : \sigma']_j^{\sigma} : \sigma$. Then the result follows by (T-Lam).

(4) (E-Pack): By inversion and IH.

(5) (E-Open): By inversion on (T-Pack), we have $\{\Delta\} \mid \Gamma \vdash e : \{\varepsilon/f\}\tau$. Then by (T-EmbedExp), we have $\{\Delta'\} \mid \Gamma \vdash [e]_j^{\tau} : \tau$. By inversion on (T-Open), $\{\Delta\} \mid \Gamma, x : \tau \vdash e' : \tau'$. Since j is fresh, e' doesn't contain any j term, so the type information from agent j doesn't affect the typing of e'. Therefore, $\{\Delta'\} \mid \Gamma, x : \tau \vdash e' : \tau'$. Finally, by substitution lemma, we have $\{\Delta'\} \mid \Gamma \vdash \{[e]_j^{\tau}/x\}e' : \tau'$.

(6) (E-EmbedPack): By inversion on (T-EmbedExp), we have $\{\Delta\} \mid \Gamma \vdash \mathsf{pack}\ (\varepsilon, v)\ \mathsf{as}\ \exists f.\ \tau : \exists f.\ \tau$, and $\{\Delta\} \mid \Gamma \vdash \exists f.\ \tau \leq_j \exists f.\ \tau'$. By type relation, we have $\tau \leq_j \tau'$. Then by inversion on (T-Pack), we have $\{\Delta\} \mid \Gamma \vdash v : \{\varepsilon/f\}\tau$. Then by T-EmbedExp, we have $\{\Delta\} \mid \Gamma \vdash [v]_j^{\{\varepsilon/f\}\tau'} : \{\varepsilon/f\}\tau'$. Then by (T-Pack), we get $\{\Delta\} \mid \Gamma \vdash \mathsf{pack}\ (\varepsilon, [v]_j^{\{\varepsilon/f\}\tau'})\ \mathsf{as}\ \exists f.\ \tau' : \exists f.\ \tau'$

□

B.2 Proof of theorem 6 (Progress for expression)

Proof. By induction on the typing rule:

(T-Pack) Let $e = \mathsf{pack}\ (\varepsilon, e')\ \mathsf{as}\ \exists f.\tau$ There are two cases. If e' is value, then we are done. If e' is not a value, by inversion and IH, we have $\langle\{\Delta\}, e'\rangle \mapsto \langle\{\Delta'\}, e''\rangle$. Then we can apply E-Pack to evaluate e.

(T-Open) Again, let $e = \mathsf{open}\ e'\ \mathsf{as}\ (f, x) \in e''$. By IH and inversion, if e' is not a value, then we can evaluate e by (E-Open2). If e' is a value, by inversion, $e' = \mathsf{pack}\ (\varepsilon, e_1)\ \mathsf{as}\ \exists f.\tau$. Then we can evaluate e by applying (E-Open1).

□